PARAMETER ADVISING FOR MULTIPLE SEQUENCE ALIGNMENT

by

Daniel Frank DeBlasio

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2016

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Daniel Frank DeBlasio, entitled Parameter Advising for Multiple Sequence Alignment and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: 15 April 2016
John Kececioglu

Date: 15 April 2016
Alon Efrat

Date: 15 April 2016
Stephen Kobourov

Date: 15 April 2016
Mike Sanderson

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: 15 April 2016
Dissertation Director: John Kececioglu

STATEMENT BY AUTHOR

ACKNOWLEDGEMENTS

The work presented in this dissertation would not have been possible without the influences of so many of the people around me. Most importantly I would not have been able to do any of the work presented here without the constant support of my advisor Dr. John Kececioglu. Dr. Kececioglu has supported me since we first met at the RECOMB meeting here in Tucson in 2009. He has both encouraged me in my research but also provided emotional (and financial) support along the way.

In addition, I would like to thank my dissertation committee Dr. Alon Efrat Dr. Stephen Kobourov and Dr. Michael Sanderson who was my advisor for my minor in Ecology and Evolutionary Biology. Dr. Kobourov in particular gave me the opportunity to work on projects in his courses that were very much outside the focus of my dissertation but were nonetheless very helpful to my comprehensive knowledge of computer science.

I would also like to acknowledge the faculty and staff that were part of the Integrative Graduate Research and Education Traineeship (IGERT) in Genomics. In particular Dr. Noah Whiteman and Dr. Matt Sullivan who allowed me to do rotations in their labs. Additionally, I would like to acknowledge Dr. Mike Barker who along with Dr. Whiteman advised me in teaching the computational component of their class. I would also like to thank the IGERT students who helped me learn to work very closely with biologists, in particular Dr. Jen Wisecaver and Andy Gloss.

The students of the Department of Computer Science also deserve a large thank you. They have all provided feedback on talks and been open to discuss ongoing problems with projects even if it was not their specific subfield.

Finally, I would like to thank all of my friends and family, who continuously provided support over the years. Most importantly they helped keep me grounded and remind me that there is a world outside the walls of Gould-Simpson.

I have received significant financial support throughout my dissertation. My initial funding came from the University of Arizona's National Science Foundation Integrative Graduate Research and Education Traineeship (IGERT) in Genomics, grant number DGE-0654435. Later funding came from NSF grant IIS-1217886 and fellowship funding from the University of Arizona Department of Computer Science. I have received several travel grants from the University of Arizona Graduate and Professional Student Council, the International Society for Computational Biology, the National Science Foundation, and Association for Computing Machinery.

I would also like to acknowledge the University of Arizona Research Computing, without an allocation of resources from High Performance Computing (HPC) and High Throughput Computing (HTC) my research would have been more difficult.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

ABSTRACT

The problem of aligning multiple protein sequences is essential to many biological analyses, but most standard formulations of the problem are NP-complete. Due to both the difficulty of the problem and its practical importance, there are many heuristic multiple sequence aligners that a researcher has at their disposal. A basic issue that frequently arises is that each of these alignment tools has a multitude of parameters that must be set, and which greatly affect the quality of the alignment produced. Most users rely on the default parameter setting that comes with the aligner, which is optimal on average, but can produce a low-quality alignment for the given inputs.

This dissertation develops an approach called *parameter advising* to find a parameter setting that produces a high-quality alignment for each given input. A parameter advisor aligns the input sequences for each choice in a collection of parameter settings, and then selects the best alignment from the resulting alignments produced. A parameter advisor has two major components: (i) an *advisor set* of parameter choices that are given to the aligner, and (ii) an *accuracy estimator* that is used to rank alignments produced by the aligner.

Alignment accuracy is measured with respect to a known reference alignment, in practice a reference alignment is not available, and we can only estimate accuracy. We develop a new accuracy estimator that we call called `Facet` (short for "feature-based accuracy estimator") that computes an accuracy estimate as a linear combination of efficiently-computable feature functions, whose coefficients are learned by solving a large scale linear programming problem. We also develop an efficient approximation algorithm for finding an advisor set of a given cardinality for a fixed estimator, whose cardinality should ideally small, as the aligner is invoked for each parameter choice in the set.

Using `Facet` for parameter advising boosts advising accuracy by almost 20% beyond using a single default parameter choice for the hardest-to-align benchmarks.

This dissertation further applies parameter advising in two ways: (i) to *ensemble alignment*, which uses the advising process on a collection of aligners to choose both the aligner and its parameter settings, and (ii) to *adaptive local realignment*, which can align different regions of the input sequences with distinct parameter choices to conform to mutation rates as they vary across the lengths of the sequences.

CHAPTER 1

Introduction and Background

Overview

While multiple sequence alignment is an essential step in many biological analyses, all of the standard formulation of the problem are NP-Complete. As a consequence, many heuristic aligners are used in practice to construct multiple sequence alignments. Each of these aligners contains a large number of tunable parameters that greatly affect the accuracy of the alignment produced. In this chapter, we introduce the concept of a *parameter advisor*, which selects a setting of parameter values for a particular set of input sequences.

1.1 Introduction

The problem of aligning a set of protein sequences is a critical step for many biological analyses, including creating phylogenetic trees, predicting secondary structure, homology modeling of tertiary structure, and many others. One issue is that while we can find optimal alignments of two sequences in polynomial time (Needleman and Wunsch, 1970), all of the standard formulations of the multiple sequence alignment problem are NP-complete (Wang and Jiang, 1994; Kececioglu and Starrett, 2004). Due to the importance of ***multiple sequence alignment*** and its complexity, it is an active field of research.

A multiple sequence alignment of set of sequences $\{S_1, S_2 ..., S_k\}$ is a k by ℓ matrix of characters, where row i in the matrix contains the characters of sequence S_i, in order, possibly with gap characters inserted. The length of the alignment ℓ is at least the length of the longest sequence, so $\ell \geq \max_{1 \leq i \leq k}\{|S_i|\}$. Characters from two sequences are said to be *aligned* when they appear in the same column of the alignment. When the two aligned characters are the same, the pair is called an *identity* otherwise it is a *mismatch*. In general, identities and mismatches are both called *substitutions*. The unweighted ***edit distance*** between two sequences is defined as the minimum number single-character edits to transform one sequence into the other. The edit value of an alignment is its total number of inserted gap characters and mismatches. For two sequences, you can find the optimal alignment of minimum value using dynamic programming (Needleman and Wunsch, 1970). Finding an optimal alignment of more than two sequences is NP-Complete (Wang and Jiang, 1994). For multiple sequence, alignment many heuristic approaches have been developed that typically use one of two common objectives. The ***sum-of-pairs*** score (SPS) of a multiple sequence alignment is the sum of the values of induced pairwise alignments Alternately, the ***tree-alignment*** objective, is the sum of pairwise alignment values align all of the branches of an input phylogenetic tree, minimized over all possible choices of ancestral sequence.

The number of alignment tools tools, or ***aligners***, available for finding multiple sequence alignments continues to grow because of the need for high quality alignments. Many methods have been published that produce multiple sequence alignments using various heuristic strategies to deal with the problem's complexity. The most popular general method is *progressive alignment* which aligns sequences using a guide tree (a binary tree where the leaves are the input sequences, Feng and Doolittle, 1987). Starting with two neighboring leaves a progressive aligner will optimally align these two sequences and replace the subtree that contained only these sequences by the alignment of the two sequences. The progressive alignment method then repeats the process proceeds in a bottom up manner aligning two of the remaining leaves (but some leaves may now contain sub-alignments). In this way a progressive aligner is only ever aligning two sequences, or alignments, to each other. This strategy has been used successfully for general alignment methods such as `Clustal` (Thompson et al., 1994), `MAFFT` (Katoh et al., 2002), `Muscle` (Edgar, 2004a,b), `Kalign` (Lassmann and Sonnhammer, 2005a), and `Opal` (Wheeler and Kececioglu, 2007). Additionally, progressive alignment strategies have also been successfully applied to specialized alignment tools such as those for whole genome alignment like `mauve` (Darling et al., 2004) those for RNA specific alignment like `PMFastR` (DeBlasio et al., 2009, 2012a) and `mLocARNA` (Will et al., 2007). Other aligners use consistency information from a library of two-sequence alignments, such as `T-Coffee` (Notredame et al., 2000), or collect sequence information into an HMM, as in `PROMALS` (Pei and Grishin, 2007). For most of the studies presented in this dissertation, we focus on the `Opal` aligner, but will later consider other aligners.

For the user, choosing an aligner is only a first step in producing a multiple sequence alignment for analysis. Each tool has a set of parameters whose values can greatly impact the quality of the computed alignment. The ***alignment parameters*** for protein sequences typically consist of the gap-open and gap-extension penalties, as well as the choice of substitution penalties for each pair of the 20 amino acids, but the available tunable parameters can differ greatly between aligners. A ***parameter choice*** for an aligner is an assignment of values to all of the alignment parameters. Work has been done (Kim and Kececioglu, 2008) to efficiently find the optimal parameter choices for an aligner that yields the highest accuracy alignments on average across a set of training benchmarks. This particular parameter choice would be the optimal ***default parameter*** choice. While such a default parameter works well in general, it can produce very low accuracy alignments for some benchmarks. Figure 1.1 shows the effect of aligning the same set of five sequences under two different alignment parameter choices, one of which is the optimal default choice.

Setting the 214 parameters for the standard protein alignment model is made easier by the fact that amino acid substitution scores are well studied. Generally one of three ***substitution matrix*** families is used for alignment: `PAM` (Dayhoff et al., 1978), `BLOSUM` (Henikoff and Henikoff, 1992), and `VTML` (Müller et al., 2002), but others also exist (Kuznetsov, 2011). Recent work has shown that the highest-accuracy alignments are generally produced using `BLOSUM` and `VTML` matrices, so these are the only ones we consider (Edgar, 2009).

We attempt to select a parameter choice that is best for a given input set of

```
d1gvoa    203 ... gsvenrarlvlevvdavcnewsad-RIGIRVSPigtfqnvdngpnee--adalyl--- ... 255
d2dora    141 ... ydfeatekllke-----vftfftk-PLGVKLPPyf---------------dlvhfdim ... 178
d1oyb     215 ... gsienrarftlevvdalveaighe-KVGLRLSPygvfnsmsggaetgivaqyayvage ... 272
d1o94a1   193 ... gslenrarfwletlekvkhavgsdcAIATRFGV-----------------dtvygpgq ... 234
d1ep3a    147 ... tdpevaaalvka-----ckavskv-PLYVKLSPnvt--------------divpiaka ... 185
```

(a) Higher Accuracy Alignment

```
d1gvoa    184 ... yl-lhqflspssnqrtdqyggsvenrarlvlevvdavcnewsad-RIGIRVSPigtfq ... 240
d2dora    159 ... kP-LGVKLPPyf--dlvhfdimaeilnqfpltYVNSV-nsig----nglfidpeaesv ... 209
d1oyb     196 ... yl-lnqfldphsntrtdeyggsienrarftlevvdalveaighe-KVGLRLSPygvfn ... 252
d1o94a1   174 ... yl-plqflnpyynkrtdkyggslenrarfwletlekvkhavgsdcAIATRF---GVdt ... 228
d1ep3a    164 ... kvPLYVKLSPnv-tdivpiakaveaagadGLTMIntl---------mgvrfdlktrqp ... 212
```

(b) Lower Accuracy Alignment

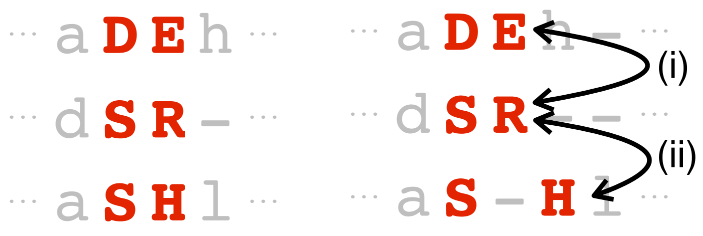
Figure 1.1: (a) Part of an alignment of benchmark sup_155 from the SABRE (Van Walle et al., 2005) suite computed by Opal (Wheeler and Kececioglu, 2007) using non-default parameter choice (VTML200, 45, 6, 40, 40); this alignment has accuracy value 75.8%, and Facet (Kececioglu and DeBlasio, 2013) estimator value 0.423. (b) Alignment of the same benchmark by Opal using the *optimal* default parameter choice (BLSM62, 65, 9, 44, 43); this alignment has lower accuracy 57.3%, and lower Facet value 0.402. In both alignments, the positions that correspond to *core blocks* of the reference alignment, which should be aligned in a correct alignment, are highlighted in bold.

sequences (rather than on average) using an approach we call ***parameter advising***, which we describe in the next section.

1.2 Parameter advising

The ***true accuracy*** of a computed alignment is measured as the fraction of substitutions that are also present in core columns of a ***reference alignment*** for these sequences. (Reference alignments represent the "correct" alignment of the sequences.) These reference alignments for protein sequences are typically constructed by aligning the three-dimensional structures of the proteins. ***Core columns*** of this reference alignment, on which we measure accuracy, are those sections where the aligned amino acids from all of the sequences are all mutually close in three-dimensional space. Figure 1.2 shows an example of computing true accuracy for a computed alignment.

What we have described and use throughout this dissertation is the sum-of-pairs



(a) Reference alignment (b) Computed alignment

Figure 1.2: A section of a reference and computed alignment. Accuracy of a computed alignment (b) is measured with respect to a known reference alignment (a). We primarily use the sum-of-pairs accuracy measure which is the fraction of aligned residues from the computed alignment recovered in the computed alignment. In the example above the aligned residue pair (i) is correctly recovered, while (ii) is not. This value is calculated only on core columns of an alignment (shown in red). In the example the accuracy is 66%, because 4 of the 6 aligned residue pairs in core columns of the reference alignment are recovered in the computed alignment.

definition of alignment accuracy. Another definition of multiple sequence alignment accuracy is known as “total-column” accuracy. The total-column accuracy is the fraction of core *columns* from the reference multiple sequence alignment that are completely recovered in the computed alignment. For the example in Figure 1.2 the sum-of-pairs accuracy is 66%, but the total-column accuracy is only 50%. Even though there is only one out of place amino acid in the alignment on the right that is from a core columns this means the whole *column* is misaligned; therefore, only one of the two core columns is recovered in the computed alignment. While arguments can be made for the merits of both the total-column and sum-of-pairs accuracy measurements, the total-column measure is more sensitive to small errors in the alignment. This is why we use the more fine-grained sum-of-pairs measure in this dissertation.

In the absence of a known reference alignment, we are left to estimate the accuracy of a computed alignment. Estimating the accuracy of a computed multiple sequence alignment (namely, how closely it agrees with the correct alignment of its sequences), without actually knowing the correct alignment, is an important problem. A good ***accuracy estimator*** has very broad utility: for example, from

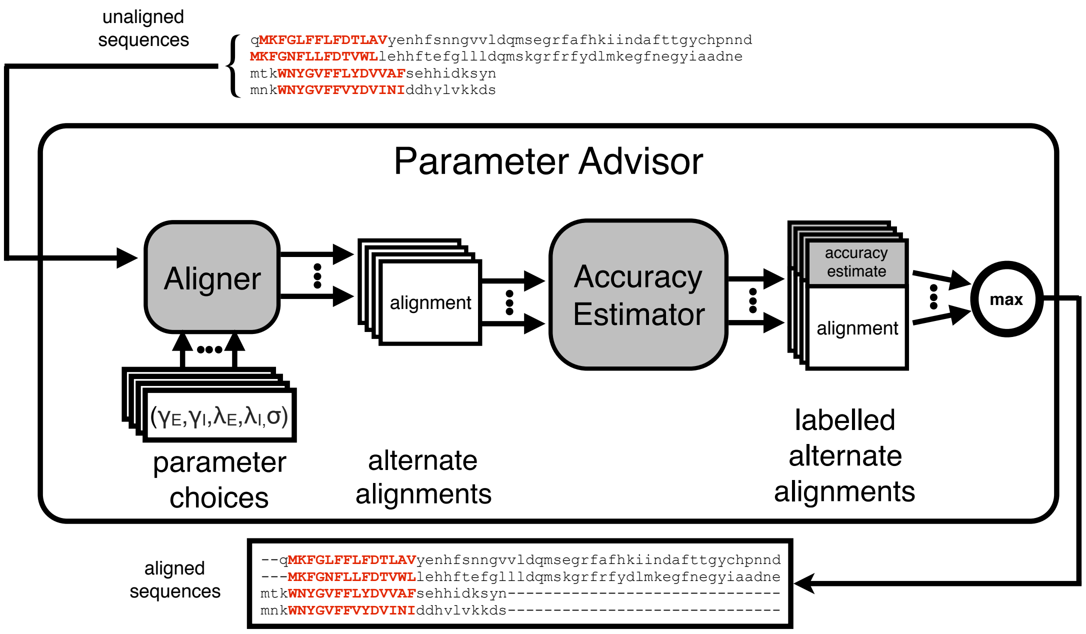

unaligned sequences
qMKFGLFFLFDTLAVyenhfsnngvvldqmsegrfafhkiindafttgychpnnd
MKFGNFLLFDTVWLlehhftefglllldqmskgrfrfydlmkegfnegyiaadne
mtkWNYGVFFLYDVVAFsehhidksyn
mnkWNYGVFFVYDVINIddhvlvkkds
Parameter Advisor
Aligner
alignment
Accuracy Estimator
accuracy estimate
alignment
max
(γE,γI,λE,λI,σ)
parameter choices
alternate alignments
labelled alternate alignments
aligned sequences
--qMKFGLFFLFDTLAVyenhfsnngvvldqmsegrfafhkiindafttgychpnnd
---MKFGNFLLFDTVWLlehhftefglllldqmskgrfrfydlmkegfnegyiaadne
mtkWNYGVFFLYDVVAFsehhidksyn-----------------------------
mnkWNYGVFFVYDVINIddhvlvkkds-----------------------------

Figure 1.3: ***Overview of the parameter advising process.*** For the `Opal` aligner a parameter choice consists of gap penalties $\gamma_E, \gamma_I, \lambda_E, \lambda_I$ as well as the substitution scoring matrix σ. A candidate alignment is generated for each parameter choice, so the advisor set should be small. An accuracy estimator labels each candidate alignment with an accuracy estimate. Finally, the alignment with the highest estimated accuracy is chosen by the advisor.

building a meta-aligner that selects the most accurate alignment output by a *collection* of aligners, to boosting the accuracy of a *single* aligner by choosing values for the parameters of its alignment scoring function to yield the best output from that aligner.

Given an accuracy estimator E, and a set P of parameter choices, a ***parameter advisor*** $\mathcal{A}$ tries each parameter choice $p \in P$, invokes an aligner to compute an alignment A_p using parameter choice p, and then "selects" the parameter choice p^* that has maximum estimated accuracy $E(A_{p^*})$. Figure 1.3 shows a diagram of the parameter advising process.

An advisor has two crucial elements:

(1) the ***advisor estimator*** which estimates the accuracy of a computed

alignment, and which the advisor will use to choose between alternate alignments, and

(2) the ***advisor set***, which is the set of parameter choices that is tried by to the aligner to produce the alternate alignments that the advisor will choose among.

We say that an advisor's accuracy on a set of input sequences is the true accuracy of the alignment obtained using the parameter choice selected from the advisor set with highest estimated accuracy.

We develop a new ***advisor estimator*** we call `Facet` (feature-based accuracy estimator) in Chapters 2 and 3. Our accuracy estimator is a linear combination of efficiently-computable feature functions. We describe the framework for the estimator and the methods for finding its coefficients in Chapter 2. We find the estimator coefficients using mathematical optimization, linear programming (LP), to identify coefficients that when used in the estimator can distinguish high accuracy alignments from low. The ***feature functions*** are measures of some aspect of an alignment that is easily computable and has a bounded value. iThe description of how to use linear programming to find an estimator as well the description of the feature functions used in `Facet` are described in Chapter 3.

To create a parameter advisor we also need to be able to find ***advisor sets*** that are of small cardinality (since the advisor will invoke the aligner for each of the parameter choices in the set) and give the best advising accuracy. An advisor set is a subset of the *parameter universe*, which is the enumeration of all possible combinations of settings for all of the parameters. We find advisor sets both for the oracle estimator (one that always returns the true accuracy of an alignment) and for a fixed estimator in Chapter 5. While finding optimal advisor sets is NP-complete, we can find optimal sets of constant cardinality in polynomial time using exhaustive search. To find advising sets of any cardinality we give a polynomial-time $\frac{\ell}{k}$***-approximation algorithm*** for finding an advisor set of cardinality k when provided an initial optimal solution of constant size $\ell < k$.

The problem of finding an ***optimal advisor*** is to simultaneously find the advisor set and advisor estimator that together yield a parameter advisor with the highest possible advising accuracy. This problem can be formulated as an integer linear program (ILP), which can be restricted to find optimal advisor sets for a fixed estimator, or an optimal advisor estimator for a fixed set. Solving the ILP is intractable in practice, even for very small training sets and using the restrictions described. Finding the optimal advisor is NP-complete (see Chapter 4), as are the problems of finding an optimal advisor, and an optimal estimator (the two restrictions to the ILP).

To learn an advisor, we collect a training set of ***example*** alignments whose true accuracy is known, and find estimator coefficients, and advising sets for the estimator, that give the best advising accuracy. We form the training set by:

(1) collecting reference alignments from standard suites of benchmark protein multiple alignments;

(2) for each such reference alignment, calling a multiple sequence aligner on the reference alignment's sequences with all parameter choices in the universe, producing a collection of alternate alignments; and

(3) labeling each such alternate alignment by its true accuracy with respect to the reference alignment for its sequences.

We use suites of benchmarks for which the reference alignments are obtained by structural alignment of the proteins using their known three-dimensional structures. The alternate alignments together with their labeled accuracies form the *examples* in our training set. Chapter 6 describes these examples in detail and experimentally demonstrates the increase in accuracy resulting from using our new advisor.

Chapters 7 and 8 show results on using the advising process for ensemble alignment (choosing both an aligner and its parameters in Chapter 7), and adaptive local realignment (realigning regions under different parameter choices in Chapter 8).

Since true accuracy is only measured on the core columns of an alignment identifying these columns could boost the accuracy of our estimator and hence our advisor.

Chapter 9 describes a method to predict how much of a column in a computed alignment is from core columns of an unknown reference alignment, using a variant of nearest neighbor classification.

Finally, Chapter 10 provides a summary of our work and future directions for research.

1.3 Survey of related work

Parameter advising as described earlier can be called ***a posteriori advising***: that is, advising on a parameter choice *after* seeing the resulting computed alignments. To our knowledge this is the first successful method for selecting alignment *parameter values* for a given input by choosing among computed alignments.

Work related to parameter advising can be divided into four major categories:

(i) ***accuracy estimation***, which attempts to provide a score for an alignment, similar to the score produced by `Facet`,

(ii) ***a priori advising*** which attempts to predict good parameter values for aligner from unaligned sequences as apposed to examining alignment accuracy after an alignment is generated,

(iii) ***meta-alignment***, which takes the output of multiple alignment methods that are known to work well, and combines together segments of those alignments, and

(iv) ***column confidence scoring***, which gives a confidence score to each column in an alignment rather than the alignment as a whole, and can be used to exclude low-confidence regions of the alignment from further analysis.

Work from each of these categories is described below.

1.3.1 Accuracy estimation

Several approaches have been developed for assessing the quality of a computed alignment without knowing a reference alignment for its sequences. These approaches follow two general strategies for estimating the accuracy with which a computed alignment recovers the unknown correct alignment.[1]

The first general strategy, which we call ***scoring-function-based***, is to develop a new scoring function on alignments that ideally is correlated with accuracy (see Notredame et al., 1998; Thompson et al., 2001; Pei and Grishin, 2001; Ahola et al., 2006, 2008). These scoring functions combine local attributes of an alignment into a score, and typically include a measure of the conservation of amino acids in alignment columns (Pei and Grishin, 2001; Ahola et al., 2006).

The second general strategy, which we call ***support-based***, is to:

(a) examine a collection of alternate alignments of the same sequences, where the collection can be generated by changing the *method* used for computing the alignment, or by changing the *input* to a method; and then

(b) measure the support for the original computed alignment among the collection of alternate alignments

(See Lassmann and Sonnhammer, 2005b; Landan and Graur, 2007; Penn et al., 2010; Kim and Ma, 2011.) In this strategy, the support for the computed alignment, which essentially measures the stability of the alignment to changes in the method or input, serves as a proxy for accuracy.

Scoring-function-based approaches

Approaches that assess alignment quality via a scoring function include `COFFEE` (Notredame et al., 1998), `AL2CO` (Pei and Grishin, 2001), `NorMD` (Thompson et al., 2001), `PredSP` (Ahola et al., 2008), and `StatSigMa` (Prakash and

[1]Here ***correctness*** can be either in terms of the unknown ***structural*** alignment (as in our present work on protein sequence alignment), or the unknown ***evolutionary*** alignment (as in simulation studies).

Tompa, 2009). Several recently developed methods also consider protein tertiary (3-dimensional) structure; due to the limitations this imposes on our benchmark set we do not compare our method with these but they include `iRMSD` (Armougom et al., 2006), `STRIKE` (Kemena et al., 2011), and an `LS-SVM` approach (Ortuño et al., 2013). We briefly describe each in turn.

`COFFEE` (Notredame et al., 1998) evaluates a multiple alignment by realigning its sequences pairwise; using the matches in all these pairwise alignments to determine transformed substitution scores for pairs of residues[2] in the columns of the multiple alignment, where these position-dependent transformed scores are in the range $[0, 1]$; accumulating the weighted sum of scores of all induced pairwise alignments in the multiple alignment without penalizing gaps, where substitutions are evaluated using the above transformed scores; and finally normalizing by the weighted sum of the lengths of all induced pairwise alignments. `COFFEE` is a component of the `T-Coffee` alignment package (Notredame et al., 2000). Updated versions of the `COFFEE` estimator have been published under a new name `TCS` (Chang et al., 2014) that use an updated library of pairwise alignments but follow the same basic principals to construct an estimation of alignment accuracy.

`AL2CO` (Pei and Grishin, 2001) uses conservation measures on alignment columns that are based on weighted frequencies of the amino acids in the column, and assesses an alignment by averaging this measure over all its columns.

`NorMD` (Thompson et al., 2001) develops an elaborate alignment scoring function that transforms standard amino acid substitution scores on pairs of aligned residues into a geometric substitution score defined on a 20-dimensional Euclidean space; takes a weighted average of all these substitution scores in a column; transforms this average substitution score through exponential decay; sums these transformed scores across columns; then includes affine gap penalties (Gotoh, 1982), and Wilbur-Lipman hash scores (Wilbur and Lipman, 1983) for normalization. `NorMD` is used in several systems, including `RASCAL` (Thompson et al., 2003), `LEON` (Thompson et al., 2004), and `AQUA` (Muller et al., 2010).

[2] A ***residue*** is a position in a protein sequence together with the amino acid at that position.

PredSP (Ahola et al., 2008) fits a beta distribution from statistics to the true accuracies associated with a sample of alignments, where the mean and variance of the distribution are transforms of a linear combination of four alignment features. The features they use are sequence percent identity, number of sequences, alignment length, and a conservation measure that is the fraction of residues in conserved columns as identified by a statistical model that takes into account amino acid background probabilities and substitution probabilities (Ahola et al., 2006). The accuracy that is predicted for the alignment is essentially the mean of the fitted beta distribution; a predicted confidence interval on the accuracy can also be quoted from the fitted distribution.

StatSigMa (Prakash and Tompa, 2009) scores an input alignment based on a phylogenetic tree, where the tree can be given by the user or based on an alignment of a second set of sequences with the same labels. A scoring function is generated based on how well the alignment fits the tree. They then test the probability of each branch in the tree given the test alignment using Karlin-Altschul (Karlin and Altschul, 1990) alignment statistics (the same statistics used for BLAST (Altschul et al., 1990) homology search). The p-value assigned to the alignment is then the maximum p-value over all branches of the tree.

iRMSD (Armougom et al., 2006) uses known tertiary structure that has been assigned to all sequences in the alignment. For each pair of sequences, and for each pair of columns, they compare the distance of the pair of columns in each protein. This difference in tertiary distances is summed and weighted to generate a score for an alignment.

STRIKE (Kemena et al., 2011) scoring uses a generated amino acid replacement matrix that scores based on how often two amino acids are in contact in the tertiary structure. The scoring algorithm infers the tertiary structure of a multiple sequence alignment from the known structure of a single protein in that alignment. They then examine the pairs of columns that are in contact (close in 3-D space) in the tertiary structure, and sum the STRIKE matrix score for each sequence's amino acid pairs at the columns in the alignment.

The LS-SVM approach (Ortuño et al., 2013) uses a similar feature-based estimator strategy to Facet. They have developed 14 feature functions for an alignment, these function each output a single numerical value and are combined to get a final accuracy estimation for an alignment. The functions used in the LS-SVM approach rely on tertiary structure, and additional information about the protein sequences obtained by querying PBD (Berman et al., 2000), PFam (Finn et al., 2009), Uniprot (Apweiler et al., 2004) and the Gene Ontology (GO, Camon et al., 2004) databases – which means these databases must be available at all times. As this method makes use of tertiary structure annotations of the proteins, it severely reduces the universe of analyzable sequences. The calculated features are fed into a least-squares support vector machine (LS-SVM) that has been trained to predict accuracy.

Support-based approaches

Approaches that assess alignment quality in terms of support from alternate alignments include MOS (Lassmann and Sonnhammer, 2005b); HoT (Landan and Graur, 2007); GUIDANCE (Penn et al., 2010); and PSAR (Kim and Ma, 2011). We briefly summarize each below.

MOS (Lassmann and Sonnhammer, 2005b) takes a computed alignment together with a collection of alternate alignments of the same sequences, and over *all* residue pairs aligned by the computed alignment, measures the *average* fraction of alternate alignments that also align the residue pair. In other words, MOS measures the average support for the substitutions in the computed alignment by other alignments in the collection.

HoT (Landan and Graur, 2007) considers a single alternate alignment, obtained by running the aligner that generated the computed alignment on the reverse of the sequences and reversing the resulting alignment, and reports the MOS value of the original alignment with respect to this alternate alignment.

GUIDANCE (Penn et al., 2010) assumes the computed alignment was generated by a so-called progressive aligner that uses a guide tree, and obtains alternate alignments by perturbing the guide tree and reinvoking the aligner. GUIDANCE reports the `MOS` value of the original alignment with respect to these alternate alignments.

`PSAR` (Kim and Ma, 2011) generates alternate alignments by probabilistically sampling pairwise alignments of each input sequence versus the pair-HMM obtained by collapsing the original alignment without the input sequence. `PSAR` reports the `MOS` value of the original alignment with respect to these alternates.

Note that in contrast to other approaches, `HoT` and `GUIDANCE` require access to the aligner that computed the alignment. They essentially measure the stability of the *aligner* to sequence reversal or guide tree alteration.

Note also that scoring-function-based approaches can estimate the accuracy of a *single* alignment, while support-based approaches inherently require a *set* of alignments.

1.3.2 ***A priori*** advising

As apposed to examining alignment accuracy after an alignment is generated, ***a priori advising*** attempts to make a prediction about an aligner's output from just the unaligned sequences. Such methods include `AlexSys` (Aniba et al., 2010), `PAcAlCI` (Ortuno et al., 2012), `GLProbs` (Ye et al., 2015), and `FSM` (Kuznetsov, 2011).

`AlexSys` (Aniba et al., 2010) uses a decision tree to classify the input sequences and identify which aligner should be used. At each step it tests the sequence's pairwise identity, sequence length differences, hydrophobicity characteristics, or `PFam` information to find relationships between sequences in the input.

`PAcAlCI` (Ortuno et al., 2012) uses similar methods to `Facet` and the `LS-SVM` approach described earlier, removing the features that rely on the alignment itself. Again features are combined using an `LS-SVM`. By querying multiple databases and finding similarity in tertiary structure, `PAcAlCI` predicts the alignment accuracy of several major alignment tools (under default parameters).

`GLProbs` (Ye et al., 2015) uses average pairwise percent-identity to determine which Hidden Markov Model (HMM) to use for alignment. While the actual difficulty assessment is simple, it then allows the HMM parameters to be specific to similarity of the particular sequences being aligned.

FSM (Kuznetsov, 2011) uses BLAST to find which family of SABmark benchmark sequences is most similar to the input sequences. It then recommends a substitution matrix that is specially tailored to these families. While BLAST is relatively fast, the applicability of this method is restricted to a narrow range of input sequences.

1.3.3 Meta-alignment

Meta-alignment uses alignments output from several aligners to construct a new alignment. Here the final alignment has aspects of the input alignments, but in contrast to other advising methods, it is not necessarily the output of any single aligner. Such methods include ComAlign (Bucka-Lassen et al., 1999), M-Coffee (Wallace et al., 2006), Crumble and Prune(Roskin et al., 2011), MergeAlign (Collingridge and Kelly, 2012), and AQUA (Muller et al., 2010).

ComAlign (Bucka-Lassen et al., 1999) identifies paths through the standard m-dimensional dynamic programming table (which in principle would yield the optimal multiple sequence alignment of the m input sequences) that corresponds to each of the candidate input multiple sequence alignments. They then find points where these paths intersect, and construct a consensus alignment by combining the highest-scoring regions of these paths.

M-Coffee (Wallace et al., 2006) uses several alignment programs to generate pairwise alignment libraries. They then use this library (rather than simply the optimal pairwise alignments) to run their T-Coffee algorithm. T-Coffee produces multiple alignments by aligning pairs of alignments to maximize the support from the library of all matching pairs of characters in each column. In this way they attempt to find the alignment with the most *support* from the other aligners.

Crumble and Prune (Roskin et al., 2011) computes large-scale alignments by splitting the input sequences both vertically (by aligning subsets of sequences) and horizontally (by aligning substrings). The Crumble procedure finds similar substrings and uses any available alignment method to align these regions; then for the overlapping regions between these blocks, it realigns these intersections to generate a full alignment. The `Prune` procedure splits an input phylogenetic tree into subproblems with a smalls number of sequences; these subproblems are then replaced by their consensus sequence when parent problems are aligned, allowing large numbers of sequences to be aligned. This method aims to reduce the computational resources needed to align large inputs, as opposed to increasing multiple sequence alignment accuracy.

`MergeAlign` (Collingridge and Kelly, 2012) generates a weighted directed acyclic graph (DAG), where each vertex represents a column in one of the input alignments, and an edge represents a transition from one column to its neighbor (the column directly to the right) in the same alignment. The weight of each edge is the number of alignments that have that transition. The consensus alignment is constructed as the maximum-weight single-source/single-sink path in the DAG.

`AQUA` (Muller et al., 2010) chooses between an alignment computed by `Muscle` (Edgar, 2004b) or `MAFFT` (Katoh et al., 2005), based on their `NorMD` (Thompson et al., 2001) score. Chapter 6 shows that for the task of choosing the more accurate alignment, the `NorMD` score used by `AQUA` is much weaker than the `Facet` estimator used here. `AQUA` can also be used as a ***meta-aligner*** because it chooses between the outputs of *multiple aligners*, rather than two parameter choices for a *singe aligner*. Chapter 7 gives results on using `Facet` in the context of meta-alignment.

1.3.4 Column confidence scoring

In addition to scoring whole alignments, work has been done to identify poorly-aligned regions of alignments. This can help biologists to find unreliable homology in an alignment to ignore for further analysis, as in programs like `GBLOCKS` (Castresana, 2000) and `ALISCORE` (Misof and Misof, 2009). Many of the accuracy estimators discussed earlier also provide column level scoring, such as `TCS`.

`GBLOCKS` (Castresana, 2000) identifies columns that are conserved and surrounded by other conserved regions, using only column percent-identity. Columns that contain gaps are eliminated, as well as runs of conservation that do not meet length cutoffs.

ALISCORE (Misof and Misof, 2009) uses a sliding window across all pairwise alignments, and determines if the window is statistically different from two random sequences. This score is evaluated on a column by counting the number of windows containing it that are significantly non-random.

Recently, Ren (2014) developed a new method to classify columns with an SVM. This method uses 5 features of an alignment that are passed into the SVM to classify whether or not the column should be used for further analysis. Their study focuses mainly on using alignments for phylogenetic reconstruction.

1.4 Review of protein secondary structure

Multiple sequence alignment benchmarks of protein sequences are normally constructed by aligning the three-dimensional structure (sometimes referred to as *tertiary structure*) of the folded proteins. Amino acids that are close in 3-D space are considered aligned, and those that are simultaneously very close in all sequences are labeled as *core columns* of the alignment. The amino acid sequence of a protein is referred to as the *primary structure.* The *secondary structure* of a protein is an intermediate between primary and tertiary structure. Secondary structure labels each amino acid as being in one of three structure classes: ***α-helix***, ***β-sheet***, or ***coil*** (other, or no structure). These structural classes tend to be conserved when the function of related proteins is conserved.

This dissertation is focused on protein multiple sequence alignment, and we exploit the fact that proteins fold into structures to perform functions within the cell when computing alignments of proteins sequences.

The tertiary structure of the proteins in a set of input sequences is not normally known, as it usually requires examining the crystalline structure of the protein, which is slow and costly. Instead we use secondary structure in our accuracy estimator, and to predict the secondary structure, we use PSIPRED (Jones, 1999).The output of PSIPRED is not only a label from the 3 secondary structure classes for each amino acid in the sequence, but also a confidence that the position in each sequence is in each structure state. We normalize the confidences so that for any amino acid the sum of the confidences for all three structure types sums to 1.

PSIPRED can make predictions using either the amino acid sequence alone, or by searching through a database of protein domains to find similar sequences using BLAST (Altschul et al., 1990). The accuracy of PSIPRED is increased substantially when using a BLAST search, so all of our results shown later are with the version of PSIPRED that searches through the UniRef90 (Suzek et al., 2007) database of protein domains (which is a non-redundant set of domains from the UniProt database (see The UniProt Consortium, 2007)) filtered using the pfilt program provided with PSIPRED.

1.5 Plan of the dissertation

Chapter 2 next describes our approach to estimating alignment accuracy as a linear combination of feature functions. It also describes how to find the coefficients for such an estimator.

Chapter 3 describes our estimator Facet (short for "feature-based accuracy estimator"). In particular, the chapter describes the efficiently computable feature functions we used in Facet.

Chapter 4 defines the problem of finding an optimal advisor (finding both the advisor set and the advisor estimator coefficients simultaneously). We also consider restrictions to finding just an optimal advisor set, or optimal advisor coefficients. We show that all of these problems are NP-complete.

Chapter 5 details the method we use to find advisor sets for a fixed estimator. While finding an optimal advisor set is NP-Complete, we have present an efficient approximation algorithm for finding near-optimal sets that perform well in practice. The chapter also describes an integer linear program for find an optimal advisor (which cannot at present be solved in practice).

Chapter 6 provides experimental results for parameter advising, and discusses the approach we use to assess the effectiveness of advising.

Chapter 7 expands the universe of parameter choices in advising to include not only the settings of the alignment parameters, but also the choice of the aligner itself which we call *aligner advising*. This yields the first true ***ensemble aligner***. We also compare the accuracy of the alignments produced by the ensemble aligner to those obtained using a parameter advisor with a fixed aligner.

Chapter 8 presents an approach called ***adaptive local realignment*** that computes alignments that can use different parameter choices in different regions of the alignment. Since regions of a protein have distinct mutation rates, using different parameter choices across an alignment can be necessary.

Chapter 9 describes an approach to predicting how much of a column in a computed alignment comes from core columns of an unknown reference alignment using a variant of nearest-neighbor classification. Since true accuracy is only measured on core columns, inferring such columns can boost the accuracy of our advisor.

Finally, Chapter 10 summarizes our results and gives future directions for research.

CHAPTER 2

Accuracy Estimation

Overview

The accuracy of a multiple sequence alignment is commonly measured as the fraction of aligned residues from the core columns of a known reference alignment that are also aligned in the computed alignment. Usually this reference alignment is unavailable, in which case we can only *estimate* the accuracy. We present a reference-free approach that estimates accuracy that is a linear combination of bounded feature functions of an alignment. In this chapter, we describe this framework for accuracy estimation and show that all higher-order polynomial estimators can be reduced to a linear estimator. We also give several approaches for learning the coefficients of the estimator function through mathematical optimization.

This chapter was adapted from portions of previous publications (DeBlasio et al., 2012b; Kececioglu and DeBlasio, 2013).

2.1 Introduction

Without knowing a reference alignment that establishes the ground truth against which the true accuracy of an alignment is measured, we are left with only being able to estimate the accuracy of an alignment. Our approach to obtaining an estimator for alignment accuracy is to (a) identify multiple *features* of an alignment that tend to be correlated with accuracy, and (b) combine these features into a single accuracy estimate. Each feature, as well as the final accuracy estimator, is a real-valued function of an alignment.

The simplest estimator is a linear combination of feature functions, where features are weighted by coefficients. These coefficients can be learned by training the estimator on example alignments whose true accuracy is known. This training process will result in a fixed coefficient or weight for each feature. Alignment accuracy is usually represented by a value in the range $[0, 1]$, with 1 corresponding to perfect accuracy. Consequently, the value of the estimator on an alignment should be bounded, no matter how long the alignment or how many sequences it aligns. For boundedness to hold when using fixed feature weights, the feature functions themselves must also be bounded. Hence, we assume that the feature functions also have the range $[0, 1]$. (The particular features we use are presented in Chapter 3.) We can then guarantee that the estimator has range $[0, 1]$ by ensuring that the coefficients found by the training process yield a convex combination of features. In practice, we have found that not all the features naturally span the entire range $[0, 1]$, so we relax the convex combination condition, and instead only require that the estimator value is in the range $[0, 1]$ on all training examples.

2.2 The estimator

In general, we consider estimators that are polynomial functions of alignment features. More precisely, suppose the features that we consider for alignments A are measured by the k feature functions $f_i(A)$ for $1 \le i \le k$. Then our accuracy estimator $E(A)$ is a polynomial in the k variables $f_i(A)$. For example, for a degree-2 polynomial,

$$E(A) \;:=\; a_0 \;+\; \sum_{1 \le i \le k} a_i\, f_i(A) \;+\; \sum_{1 \le i,j \le k} a_{ij}\, f_i(A)\, f_j(A).$$

For a polynomial of degree d, our *accuracy estimator* $E(A)$ has the general form,

$$E(A) \;:=\; \sum_{\substack{p_1,\ldots,p_k \,\in\, \mathcal{Z}^+ \\ p_1+\cdots+p_k \,\le\, d}} a_{p_1,\ldots,p_k} \prod_{1 \le i \le k} \big(f_i(A)\big)^{p_i},$$

where $\mathcal{Z}^+$ denotes the nonnegative integers, and the coefficients on the terms of the polynomial are given by the $a_{p_1,\ldots,p_k}$. In this summation, there are k index variables p_i, and each possible *assignment* of nonnegative integers to the p_i that satisfies $\sum_i p_i \leq d$ specifies one *term* of the summation, and hence the powers for one term of the polynomial.

Encoding higher-order polynomial estimators

Learning an estimator from example alignments, as discussed in Section 2.3, corresponds to determining the coefficients for its terms. We can efficiently learn optimal values for the coefficients, that minimize the error between the estimate $E(A)$ and the actual accuracy of alignment A on a set of training examples, even for estimators that are polynomials of *arbitrary* degree d. This can be done for arbitrary degree essentially because such an estimator can always be reduced to the linear case by a change of feature functions, as follows. For *each* term in the degree-d estimator, where the term is specified by the powers p_i of the f_i, define a *new* feature function

$$g_j(A) \;\; := \prod_{1 \leq i \leq k} \big(f_i(A)\big)^{p_i},$$

that has an associated coefficient $c_j := a_{p_1,\ldots,p_k}$. Then in terms of the new feature functions g_j, the original degree-d estimator is equivalent to the linear estimator

$$E(A) \;\; = \;\; c_0 + \sum_{1 \leq j < t} c_j \, g_j(A),$$

where t is the number of terms in the original polynomial. For a degree-d estimator with k original feature functions, the number of coefficients t in the linearized estimator is at least $\mathcal{P}(d, k)$, the number of integer partitions of d with k parts. This number of coefficients grows very fast with d, so overfitting can become an issue when learning a high-degree estimator. (Even a cubic estimator on 10 features already has 286 coefficients.) In our experiments, we focus on *linear* estimators.

The coefficients of the estimator polynomial are found by mathematical optimization which we will describe next.

2.3 Learning the estimator from examples

In Section 1.2 we described the set of ***examples***: benchmark sequences that have been aligned under various parameter choices by the aligner, and whose alignment are labeled with their true accuracy. In addition, we record the feature function values for each of these examples. We then use these examples, with their associated accuracy and feature values, to find coefficients that fit the accuracy estimator to true accuracy by two techniques that we describe below.

2.3.1 Fitting to accuracy values

A natural criterion for fitting the estimator is to minimize the error on the example alignments between the estimator and the true accuracy value. For alignment A in our training set $\mathcal{S}$, let $E_c(A)$ be its estimated accuracy where vector $c = (c_0, \ldots, c_{t-1})$ specifies the values for the coefficients of the estimator polynomial, and let $F(A)$ be the *true accuracy* of example A.

Formally, minimizing the weighted error between estimated accuracy and true accuracy yields estimator $E^* := E_{c^*}$ with coefficient vector

$$c^* \;\; := \;\; \operatorname*{argmin}_{c \,\in\, \mathcal{R}^t} \; \sum_{A \,\in\, \mathcal{S}} w_A \, \big| E_c(A) - F(A) \big|^p,$$

where power p controls the degree to which large accuracy errors are penalized. Weights w_A correct for sampling bias among the examples, as explained below.

When $p = 2$, this corresponds to minimizing the L_2 norm between the estimator and the true accuracies. The absolute value in the objective function may be removed, and the formulation becomes a *quadratic programming problem* in variables c, which can be efficiently solved. (Note that E_c is linear in c.) When $p = 1$, the formulation corresponds to minimizing the L_1 norm. This is less sensitive to outliers than the L_2 norm, which can be advantageous when the underlying features are noisy. Minimizing the L_1 norm can be reduced to a *linear programming problem* as follows. In addition to variables c, we have a second vector of variables e with an entry e_A for each example $A \in \mathcal{S}$ to capture the absolute value in the L_1 norm, along with the inequalities,

$$\begin{aligned} e_A &\geq E_c(A) - F(A), \\ e_A &\geq F(A) - E_c(A), \end{aligned}$$

which are linear in variables c and e. We then minimize the linear objective function

$$\sum_{A \in \mathcal{S}} w_A \, e_A.$$

For n examples, the linear program has $n + t$ variables and $O(n)$ inequalities, which is solvable even for very large numbers of examples.

If the feature functions all have range $[0, 1]$, we can ensure that the resulting estimator E^* also has range $[0, 1]$ by adding to the the linear inequalities,

$$\begin{aligned} c_i &\geq 0, \\ \sum_{0 \leq i < t} c_i &\leq 1. \end{aligned}$$

But as mentioned earlier, it may be useful to not restrict the coefficients to be a convex combination because while the features are bounded, they may not have values across the whole range. Instead we can also add the following inequalities for each training example A that ensure E^* has range $[0, 1]$.

$$\begin{aligned} E_c(A) &\geq 0, \\ E_c(A) &\leq 1. \end{aligned}$$

The weights w_A on examples aid in finding an estimator that is good across all accuracies. In the suites of protein alignment benchmarks that are commonly available, a predominance of the benchmarks consist of sequences that are easily alignable, meaning that standard aligners find high-accuracy alignments for these benchmarks.[1] In this situation, when training set $\mathcal{S}$ is generated as described earlier,

[1]This is mainly a consequence of the fact that proteins for which reliable structural reference alignments are available tend to be closely related, and hence easier to align. It does not mean that typical biological inputs are easy.

most examples have high accuracy, with relatively few at moderate to low accuracies. Without weights on examples, the resulting estimator E^* is strongly biased towards optimizing the fit for high accuracy alignments, at the expense of a poor fit at lower accuracies. To prevent this, we bin the examples in $\mathcal{S}$ by their true accuracy, where $\mathcal{B}(A) \subseteq \mathcal{S}$ is the set of alignments falling in the bin for example A, and then weight the error term for A by $w_A := 1/\big|\mathcal{B}(A)\big|$. (In our experiments, we form 10 bins equally spaced at 10% increments in accuracy.) In the objective function this weights *bins* uniformly (rather than weighting *examples* uniformly) and weights the error equally across the full range of accuracies.

2.3.2 Fitting to accuracy differences

Many applications of an accuracy estimator E will use it to choose from a set of alignments the one that is estimated to be most accurate. (This occurs, for instance, in parameter advising as discussed in Chapter 6.) In such applications, the estimator is effectively ranking alignments, and all that is needed is for the estimator to be *monotonic* in true accuracy. Accordingly, rather than trying to fit the estimator to match accuracy *values*, we can instead fit it so that *differences* in accuracy are reflected by at least as large differences in the estimator. This fitting to differences is less constraining than fitting to values, and hence might be better achieved.

More precisely, suppose we have selected a set $\mathcal{P} \subseteq \mathcal{S}^2$ of ordered pairs of example alignments, where every pair $(A, B) \in \mathcal{P}$ satsifies $F(A) < F(B)$. Set $\mathcal{P}$ holds pairs of examples on which accuracy F increases for which we desire similar behavior from our estimator E. (Later we discuss how we select a small set $\mathcal{P}$ of important pairs.) If estimator E increases at least as much as accuracy F on a pair in $\mathcal{P}$, this is a success, and if it increases less than F, we consider the amount it falls short an error, which we try to minimize. Notice this tries to match large accuracy increases, and penalizes less for not matching small increases.

We formulate fitting to differences as finding the optimal estimator $E^* := E_{c^*}$

given by coefficients

$$c^* \;:=\; \operatorname*{argmin}_{c \in \mathcal{R}^t} \sum_{(A,B)\,\in\,\mathcal{P}} w_{AB} \left(\max\Big\{ \big(F(B) - F(A)\big) \;-\; \big(E_c(B) - E_c(A)\big),\; 0 \Big\} \right)^p,$$

where w_{AB} weights the error term for a pair. When power p is 1 or 2, we can reduce this optimization problem to a linear or quadratic program as follows. We introduce a vector of variables e with an entry e_{AB} for each pair $(A, B) \in \mathcal{P}$, along with the inequalities,

$$\begin{aligned} e_{AB} &\;\geq\; 0, \\ e_{AB} &\;\geq\; \big(F(B) - F(A)\big) \;-\; \big(E_c(B) - E_c(A)\big), \end{aligned}$$

which are linear in variables c and e. We then minimize the objective function,

$$\sum_{(A,B)\in\mathcal{P}} w_{AB}\, (e_{AB})^p,$$

which is linear or quadratic in the variables for $p = 1$ or 2.

For a set $\mathcal{P}$ of m pairs, these programs have $m + t$ variables and m inequalities, where $m = O(n^2)$ in terms of the number of examples n. For the programs to be manageable for large n, set $\mathcal{P}$ must be quite sparse.

We can select a sparse set $\mathcal{P}$ of important pairs using one of two methods: threshold-minimum accuracy difference pairs, or distributed-example pairs. Recall that the training set $\mathcal{S}$ of examples consists of alternate alignments of the sequences in benchmark reference alignments, where the alternates are generated by aligning the benchmark under a constant number of different parameter choices.

Threshold-difference pairs

While we would like an accuracy estimator that matches the difference in true accuracy between *any* two alignments, in parameter advising we are only concerned with choosing among alignments over the *same* sets of sequences. With threshold-difference pairs, we include in $\mathcal{P}$ only pairs of alignments (A, B) of the *same benchmark*. In particular, we include all such pairs where $F(A) - F(B) \geq \epsilon$. Here $\epsilon > 0$

is a tunable threshold; if the difference in accuracy is smaller than this threshold, we exclude it from training, as its effect on the parameter advisor is minimal, and it makes the linear or quadratic problem much harder to solve. As ϵ approaches 0, the better the estimator will be at distinguishing small differences, but more constraints will be included in the program increasing the running time of the solver. For example pairs under this model, we set the weight w_{AB} to be $\frac{1}{|\mathcal{B}(C)|}$, where $\mathcal{B}$ gives the corresponding bin for benchmarks aligned under the default parameter settings, and C is the alignment under the default parameter settings of the benchmark sequences that A and B are aligning.

Distributed-example pairs

An estimator that is designed for parameter advising should rank the highest accuracy alternate alignment for a benchmark above the other alternates for that benchmark. Consequently, for each benchmark we select for $\mathcal{P}$ its highest-accuracy alternate, paired with its other alternates for which their difference in accuracy is at least ϵ, where ϵ is a tunable threshold. (Notice this picks $O(n)$ pairs on the n examples.) For the estimator to generalize beyond the training set, it helps to also properly rank alignments between benchmarks. To include some pairs between benchmarks, we choose the minimum, maximum, and median accuracy alignments for each benchmark, and form one list L of all these chosen alignments, ordered by increasing accuracy. Then for each alignment A in L, we scan L to the right to select the first k pairs (A, B) for which $F(B) \geq F(A) + i\,\delta$ where $i = 1, \ldots, k$, and for which B is from a different benchmark than A. While the constants $\epsilon \geq 0$, $\delta \geq 0$, and $k \geq 1$ control the specific pairs that this procedure selects for $\mathcal{P}$, it always selects $O(n)$ pairs on the n examples.

Weighting distributed-example pairs

When fitting to accuracy differences, we again weight the error terms, which are now associated with pairs, to correct for sampling bias within $\mathcal{P}$. We want the weighted pairs to treat the entire accuracy range equally, so the fitted estimator performs well at all accuracies. As when fitting to accuracy values, we partition the example alignments in $\mathcal{S}$ into bins $\mathcal{B}_1, \ldots, \mathcal{B}_k$ according to their true accuracy. To model equal weighting of accuracy bins by pairs, we consider a pair $(A, B) \in \mathcal{P}$ to have half its weight w_{AB} on the bin containing A, and half on the bin containing B. (So in this model, a pair (A, B) with both ends A, B in the same bin $\mathcal{B}$, places all its weight w_{AB} on $\mathcal{B}$.) Then we want to find weights $w_{AB} > 0$ that, for all bins $\mathcal{B}$, satisfy

$$\sum_{(A,B)\,\in\,\mathcal{P}\,:\,A\,\in\,\mathcal{B}} \tfrac{1}{2}\, w_{AB} \;+\; \sum_{(A,B)\,\in\,\mathcal{P}\,:\,B\,\in\,\mathcal{B}} \tfrac{1}{2}\, w_{AB} \;=\; 1.$$

In other words, the pairs should weight bins uniformly.

We say a collection of weights w_{AB} are *balanced* if they satisfy the above property on all bins $\mathcal{B}$. While balanced weights do not always exist in general, we can identify an easily-satisfied condition that guarantees they do exist, and in this case find balanced weights by the following graph algorithm.

Construct an undirected graph G whose vertices are the bins $\mathcal{B}_i$ and whose edges (i, j) go between bins $\mathcal{B}_i, \mathcal{B}_j$ that have an alignment pair (A, B) in $\mathcal{P}$ with $A \in \mathcal{B}_i$ and $B \in \mathcal{B}_j$. (Notice G has self-loops when pairs have both alignments in the same bin.) Our algorithm first computes weights ω_{ij} on the edges (i, j) in G, and then assigns weights to pairs (A, B) in $\mathcal{P}$ by setting $w_{AB} := 2\,\omega_{ij}/c_{ij}$, where bins $\mathcal{B}_i, \mathcal{B}_j$ contain alignments A, B, and c_{ij} counts the number of pairs in $\mathcal{P}$ between bins $\mathcal{B}_i$ and $\mathcal{B}_j$. (The factor of 2 is due to a pair only contributing weight $\frac{1}{2} w_{AB}$ to a bin.) A consequence is that all pairs (A, B) that go between the same bins get the same weight w_{AB}.

During the algorithm, an edge (i, j) in G is said to be *labeled* if its weight ω_{ij} has been determined; otherwise it is *unlabeled*. We call the *degree* of a vertex i the total number of endpoints of edges in G that touch i, where a self-loop contributes two endpoints to the degree. Initially all edges of G are unlabeled. The algorithm sorts the vertices of G in order of nonincreasing degree, and then processes the vertices from highest degree on down.

In the general step, the algorithm processes vertex i as follows. It accumulates w, the sum of the weights ω_{ij} of all *labeled* edges that touch i; counts u, the number of *unlabeled* edges touching i that are not a self-loop; and determines d, the degree of i. To the unlabeled edges (i,j) touching i, the algorithm assigns weight $\omega_{ij} := 1/d$ if the edge is not a self-loop, and weight $\omega_{ii} := \frac{1}{2}(1 - w - \frac{u}{d})$ otherwise.

This algorithm assigns *balanced weights* if in graph G, every bin has a self-loop, as stated in the following theorem.

Theorem 1 (Finding Balanced Weights) *Suppose every bin $\mathcal{B}$ has some pair (A,B) in $\mathcal{P}$ with both alignments A,B in $\mathcal{B}$. Then the above graph algorithm finds balanced weights.*

Proof We will show that: (a) for every edge (i,j) in G, its assigned weight satisfies $\omega_{ij} > 0$; and (b) for every vertex i, the weights assigned to its incident edges (i,j) satisfy

$$\sum_{(i,j)\,:\,j\neq i} \omega_{ij} \;+\; 2\,\omega_{ii} \;=\; 1.$$

From properties (a) and (b) it follows that the resulting weights w_{AB} are balanced.

The key observation is that when processing a vertex i of degree d, the edges touching i that are already *labeled* will have been assigned a weight $\omega_{ij} \leq 1/d$, since the other endpoint j must have degree at least d (as vertices are processed from highest degree on down). Unlabeled edges touching i, other than a self-loop, get assigned weight $\omega_{ij} = 1/d > 0$. When assigning weight ω_{ii} for the unlabeled self-loop, the total weight w of incident labeled edges satisfies $w \leq (d-u-2)/d$, by the key observation above and the fact that vertex i always has a self-loop which contributes 2 to its degree. This inequality in turn implies $\omega_{ii} \geq 1/d > 0$. Thus property (a) holds.

Furthermore, twice the weight ω_{ii} assigned to the self-loop takes up the slack between 1 and the weights of all other incident edges, so property (b) holds as well.

□

Regarding the condition in Theorem 1, if there are bins without self-loops, balanced weights do not necessarily exist. The smallest such instance is when G is a path of length 2.

Notice that we can ensure the condition in Theorem 1 holds if every bin has at least two example alignments: simply add a pair (A, B) to $\mathcal{P}$ where both alignments are in the bin, if the procedure for selecting a sparse $\mathcal{P}$ did not already. When the training set $\mathcal{S}$ of example alignments is sufficiently large compared to the number of bins (which is within our control), every bin is likely to have at least two examples. So Theorem 1 essentially guarantees that in practice we can fit our estimator using balanced weights.

For k bins and m pairs, the pair-weighting algorithm can be implemented to run in $O(k + m)$ time, using radix sort to map pairs in $\mathcal{P}$ to edges in G, and counting sort to order the vertices of G by degree.

Summary

In this chapter, we have developed an accuracy estimator that is a linear combination of feature functions, and provided two approaches to learning the coefficients of this estimator. Chapter 3 next describes the specific features that along with this framework make up the `Facet` accuracy estimator. Results on using the `Facet` estimator with the feature functions described in the next chapter are presented in Chapter 6.

CHAPTER 3

The `Facet` Estimator

Overview

In Chapter 2, we described a general framework for creating an alignment accuracy estimator that is a linear combination of feature functions, and for learning the coefficients of such an estimator. In this chapter, we explore the feature functions used in our accuracy estimator `Facet`. Some of the features we use are standard metrics that are common for measuring multiple sequence alignment quality, such as amino acid percent identity and gap extension density, but many of the most reliable features are novel. The strongest feature functions tend to use predicted secondary structure. We describe in detail the most accurate and novel features: secondary structure blockiness and secondary structure consensus.

This chapter was adapted from portions of previous publications (DeBlasio et al., 2012b; Kececioglu and DeBlasio, 2013).

3.1 Introduction

In Section 1.3.1 we described two classes of accuracy estimators: *scoring-function-based* and *support-based*. While our approach is within the general scoring-function-based category, compared to prior such approaches, we:

(a) introduce several *novel feature functions* that measure non-local properties of an alignment and have stronger correlation with accuracy (such as Secondary Structure Blockiness, described here in Section 3.2.1),

(b) consider *larger classes of estimators* beyond linear combinations of features (such as quadratic polynomials, described in Chapter 2), and

(c) develop *new regression formulations* for learning an estimator from examples (such as difference fitting, described in Chapter 2).

Our approach can readily incorporate new feature functions into the estimator, and is easily tailored to a particular class of alignments by choosing appropriate features and performing regression.

Compared to support-based approaches, our estimator does not degrade on difficult alignment instances, where for parameter advising, good accuracy estimation can have the greatest impact. As shown in our advising experiments in Chapter 6, support-based approaches lose the ability to detect accurate alignments of hard-to-align sequences, since for such sequences most alternate alignments are poor and lend little support to the alignment that is actually most accurate.

In this chapter, we begin by giving descriptions of the feature functions used in the `Facet` estimator in Section 3.2. For each feature, we also consider a few variants.In the next section we discus the most accurate feature function, called Secondary Structure Blockiness. Section 3.2.13 shows examples of the feature values for a set of computed alignments. Section 3.3 details our implementation of `Facet` in `Java`. Not only is `Facet` available as a stand alone tool that can be incorporated into existing analysis pipelines that include multiple sequence alignment, it can also be used via an API within other multiple sequence alignment tools.

3.2 Estimator features

The quality of the estimator that results from our approach ultimately rests on the quality of the features that we consider. We consider twelve features of an alignment, the majority of which are novel. All are efficiently computable, so the resulting estimator is fast to evaluate. The strongest feature functions make use of predicted *secondary structure* (which is not surprising, given that protein sequence alignments are often surrogates for structural alignments). Details about protein secondary structure and, how we predict it for new proteins, can be found in Section 1.4.

Another aspect of some of the best alignment features is that they tend to use *non-local information*. This is in contrast to standard ways of scoring sequence alignments, such as with amino acid substitution scores or gap open and extension penalties, which are often a function of a single alignment column or two adjacent columns (as is necessary for efficient dynamic programming algorithms). While a good accuracy estimator would make an ideal scoring function for *constructing* a sequence alignment, computing an optimal alignment under such a nonlocal scoring function seems prohibitive (especially since multiple alignment is already NP-complete for the current highly-local scoring functions). Nevertheless, given that our estimator can be efficiently evaluated on any constructed alignment, it is well suited for *selecting* a sequence alignment from among several alternate alignments, as we discuss in Chapter 6 in the context of parameter advising (and later chapters further consider the contexts of ensemble alignment and adaptive local realignment).

Key properties of a good feature function are: (a) it should measure some attribute that discriminates high accuracy alignments from others, (b) it should be efficiently computable, and (c) its value should be bounded (as discussed at the beginning of Chapter 2). Bounded functions are easily normalized, and we scale all our feature functions to the range $[0, 1]$. We also intend our features to be increasing functions of, or positively correlated with, alignment accuracy.

The following are the alignment feature functions we consider for our accuracy estimator. We highlight the first function as it is the most novel, one of the strongest, and is the most challenging to compute.

3.2.1 Secondary Structure Blockiness

The reference alignments in the most reliable suites of protein alignment benchmarks are computed by structural alignment of the known three-dimensional structures of the proteins. The so-called *core blocks* of these reference alignments, which are the columns in the reference to which an alternate alignment is compared when measuring its true accuracy, are typically defined as the regions of the structural alignment in which the residues of the different proteins are all within a small distance threshold of each other in the superimposed structures. These regions of structural agreement are usually in the embedded core of the folded proteins, and the secondary structure of the core usually consists of α-helices and β-strands. (Details of *secondary structure* and its representation can be found in Section 1.4.) As a consequence, in the reference sequence alignment, the sequences in a core block often share the same secondary structure, and the type of this structure is usually α-helix or β-strand.

We measure the degree to which a multiple alignment displays this pattern of structure by a feature we call *Secondary Structure Blockiness*. Suppose that for the protein sequences in a multiple alignment we have predicted the secondary structure of each protein, using a standard prediction tool such as `PSIPRED` (Jones, 1999). Then in multiple sequence alignment A and for given integers $k, \ell > 1$, define a *secondary structure block* $\mathcal{B}$ to be:

(i) a contiguous interval of at least ℓ columns of A, together with

(ii) a subset of at least k sequences in A, such that on all columns in this interval, in all sequences in this subset, all the entries in these columns for these sequences have the same predicted secondary structure type, and this shared type is all α-helix or all β-strand.

We call $\mathcal{B}$ an α-block or a β-block according to the common type of its entries. Parameter ℓ, which controls the minimum width of a block, relates to the minimum length of α-helices and β-strands; we can extend the definition to use different values ℓ_α and ℓ_β for α- and β-blocks.

A *packing* for alignment A is a set $\mathcal{P} = \{\mathcal{B}_1, \ldots, \mathcal{B}_b\}$ of secondary structure blocks of A, such that the column intervals of the $\mathcal{B}_i \in \mathcal{P}$ are all disjoint. (In other words, in a packing, each column of A is in at most one block. The sequence subsets for the blocks can differ arbitrarily.) The *value* of a block is the total number of residue pairs (or equivalently, substitutions) in its columns; the value of a packing is the sum of the values of its blocks.

Finally, the *blockiness* of an alignment A is the maximum value of any packing for A, divided by the total number of residue pairs in the columns of A. In other words, Secondary Structure Blockiness measures the fraction of substitutions in A

that are in an optimal packing of α- or β-blocks.

At first glance measuring blockiness might seem hard (since optimal packing problems are often computationally intractable), yet surprisingly it can actually be computed in *linear time* in the size of the alignment, as the following theorem states. The main idea is that evaluating blockiness can be reduced to solving a longest path problem on a directed acyclic graph of linear size.

Theorem 2 (Evaluating Blockiness) *Given a multiple alignment A of m protein sequences and n columns, where the sequences are annotated with predicted secondary structure, the blockiness of A can be computed in $O(mn)$ time.*

Proof The key is to not enumerate subsets of sequences in A when considering blocks for packings, and instead enumerate intervals of columns of A. Given a candidate column interval I for a block $\mathcal{B}$, we can avoid considering all possible subsets of sequences, since there are only two possibilities for the secondary structure type s of $\mathcal{B}$, and the sequences in $\mathcal{B}$ must have type s across I. To maximize the *value* of $\mathcal{B}$, we can collect all sequences in A that have type α across I (if any), all sequences that have type β across I, and keep whichever subset has more sequences.

Following this idea, given alignment A, we form an edge-weighted, directed graph G that has a vertex for every column of A, plus an artificial *sink* vertex, and an edge of weight 0 from each column to its immediate successor, plus an edge of weight 0 from the last column of A to the sink. We call the vertex for the first column of A the *source* vertex. We could then consider *all* intervals I of at least ℓ columns, test whether the best sequence subset for each I as described above has at least k sequences, and if so, add an edge to G from the first column of I to the immediate successor of the last column of I, weighted by the maximum value of a block with interval I. A *longest path* in the resulting graph G from its source to its sink then gives an optimal packing for A, and the blockiness of A is the length of this longest path divided by the total number of substitutions in A. This graph G would have $\Theta(n^2)$ edges, however, and would not lead to an $O(mn)$ time algorithm for blockiness. Instead, we *only* add edges to G for such intervals I whose number of columns, or *width*, is in the range $[\ell, 2\ell - 1]$. Any block $\mathcal{B}$ whose interval has width at least ℓ is the concatenation of disjoint blocks whose intervals have widths in the above range. Furthermore, the value of block $\mathcal{B}$ is the sum of the values of the blocks in the concatenation. Only adding to G edges in the above width range gives a sparse graph with $O(n)$ vertices and just $O(\ell n)$ edges, which is $O(n)$ edges for constant ℓ.

To implement this algorithm, first construct G in $O(mn)$ time by (1) enumerating the $O(n)$ edges of G in lexicographic order on the pair of column indices defining the column interval for the edge, and then (2) determining the weight of each successive edge e in this order in $O(m)$ time by appending a single column of A to form the column interval for e from the shorter interval of its predecessor. Graph G is acyclic, and a longest source-sink path in a directed acyclic graph can be computed in time linear in its number of vertices and edges (Cormen et al., 2009, pp. 655–657) so the optimal packing in A by blocks can be obtained from G in $O(n)$ time. This takes $O(mn)$ time in total. $\square$

There are a few further details in how we use Secondary Structure Blockiness in practice which are discussed below.

(a) Blocks are calculated first the on structure classes α-helix and β-strand. We have an option to then also construct coil blocks on the columns of an alignment that are not already covered by α-helix and β-strand blocks. In practice, we found that including this second coil pass increases the advising accuracy over only including blocks for non-coil classes.

(b) We also specify a minimum number of rows k in the definition of a block. We fine that in practice, blockiness shows the best performance when this minimum is set to $k = 2$ rows. While using a minimum of $k = 1$ would not have affected the results if we only used $\alpha-$ or $\beta-$blocks, using $k > 1$ increased the number of columns that could be included in coil blocks.

(c) Permitting gap characters in blocks allows them to be extended to regions that may have single insertions or deletions in one or more sequences. When gaps are allowed in a block they do not contribute to the value of the block (as the value is still defined as the number of residue pairs in the columns and rows of the block), but they can extend a block to include more rows. We find that including gaps increases the accuracy advising with of blockiness in practice.

(d) In reality, α-helix and β-strand physically both have a minimum number of amino acids to form their structures. We have two modes to capture this: one that sets the minimum based on actual physical sizes and one that sets the minimums to the same length. In the unequal mode, the minimum sizes α-helix, $\ell_\alpha = 4$; β-strand, $\ell_\beta = 3$; and coil, $\ell_c = 2$. In equal mode, $\ell_\alpha = \ell_\beta = \ell_c = 2$. We find that in practice, the unequal mode gives better advising accuracy.

(e) The secondary structure prediction tool `PSIPRED` outputs confidence values p_t for each structure type $t \in \{\alpha, \beta, c\}$. These can be used to choose a single structure prediction at each position in a protein, by assigning the prediction with the highest confidence value. Alternately, we can choose a threshold τ, and say that a residue in the protein has not just one structure type, but all structure types with $p_t > \tau$. In this way, residues can be in multiple blocks of different structure types if both types have high confidence; in the final packing however, it will only be in one since the blocks of a packing are column-disjoint. We found that in practice, using confidences in this way to allow ambiguous structural types was detrimental to advising accuracy on the benchmarks we considered.

The remaining feature functions in `Facet` are simpler to compute than Secondary Structure Blockiness.

3.2.2 Secondary Structure Agreement

The secondary structure prediction tool `PSIPRED` (Jones, 1999) outputs confidence values at each residue that are intended to reflect the probability that the residue has each of the three secondary structure types. Denote these three confidences for a residue i, r (the residue in the i-th sequence at the r-th column), normalized so they add up to 1, by $p_\alpha(i,r)$, $p_\beta(i,r)$, and $p_\gamma(i,r)$. Then we can estimate the probability that two sequences i, j in column r have the same secondary structure type that is not coil, by looking at the support for that pair from all intermediate sequences k. We first define the similarity of two residues (i,r) and (j,r) in column r as

$$S(i,k,r) \ := \ p_\alpha(i,r)\, p_\alpha(k,r) \ + \ p_\beta(i,r)\, p_\beta(k,r).$$

To measure how strongly the secondary structure locally agrees between sequences i and j around column r, we compute a weighted average P of S in a window of width $2\ell + 1$ centered around column r,

$$P(i,j,r) \ := \sum_{-\ell \le p \le \ell} w_p \, S(k,j,r+p)$$

where the weights w_p form a discrete distribution that peaks at $p = 0$ and is symmetric.

We can define the support for the pair i, j from intermediate sequence k as the product of the similarities of each i and j with k, $P(i,k,r)\, P(k,j,r)$. The support Q for pair i, j from all intermediate sequences is then defined as

$$Q(i,j,r) \ := \sum_{\substack{0 \le k \le N \\ i \ne k \\ j \ne k}} P(i,k,r)\ P(k,j,r),$$

The value of the Secondary Structure Agreement feature is then the average of $Q(i,j,r)$ over all sequence pairs i, j in all columns r.

This is the feature with the largest running time, but is also one of the strongest features. Is running time is $O(m^3 n \ell)$ for m sequences in an alignment of length n.

The value of ℓ and w must be set by the user. We tried various values for both, and found that $\ell = 2$ and $w = (0.7, 0.24, 0.28, 0.24, 0.7)$ gave the best advising results.

3.2.3 Gap Coil Density

A *gap* in a pairwise alignment is a maximal run of either insertions or deletions. For every pair of sequences, there is a set of *gap-residue pairs* (residues that are aligned with gap characters) which each has an associated secondary structure prediction given by `PSIPRED` (the structure assigned to the residue in the pair). The Gap Coil Density feature measures the fraction of all gap-residue pairs with a secondary structure type of *coil.*

As described, computing Gap Coil Density may seem quadratic in the number of sequences. By simply counting the number of gaps g_i, coil-labeled non-gap entries γ_i, and non-coil-labeled non-gap s_i entries in column i, we can compute this feature by

$$\frac{\displaystyle\sum_{\text{columns } i} g_i \, \gamma_i}{\displaystyle\sum_{\text{columns } i} g_i \, (\gamma_i + s_i)}.$$

All this counting takes linear time total in the number of sequences, so the running time for computing Gap Coil Density is $O(mn)$.

Alternately, we can use `PSIPRED` confidences; the feature value is then the *average* coil confidences over all gap-residue pairs in the alignment. We find that in practice, using these confidences gives better advising accuracy.

3.2.4 Gap Extension Density

This feature counts the number of *null characters* in the alignment (the dashes that denote gaps). This is related to affine gap penalties (Gotoh 1982), which are commonly used to score alignments. We normalize this count by the total number of alignment entries, or an upper bound U on the number of possible entries. The reason to use the upper bound is so that we can compare the feature value across alignments of the same sequences that may have different alignment lengths, while still yielding a feature value that lies in the range $[0, 1]$. We calculate this upper bound as

$$U \;:=\; \binom{N}{2}\left(\max_{s\in S}|s| \;+\; \max_{s'\in S'}|s'|\right),$$

where $S' \;:=\; S - \mathrm{argmax}_{s\in S}\,|s|$, ls that the second max gives the length of the second longest sequence in S. We find that normalizing by U gives better advising accuracies.

As the quantity described above is generally *decreasing* in to alignment accuracy (since more gaps generally indicates a lower quality alignment), for the actual feature value we use 1 minus this ratio described above.

Gap Extension Density essentially counts the number of *null characters* in an alignment, which can be done in linear time for each sequence. Thus Gap Extension Density can be computed in $O(mn)$ time. The lengths of the input sequences can be computed in linear, so U can be computed in this same amount of time.

3.2.5 Gap Open Density

This feature counts the number of *runs* of null characters in the rows of the alignment (which also relates to affine gap penalties). Again we provide options to normalize by the total length of all such runs, or by upper-bound U of alignment size (which tends to give better advising accuracy). Just as with Gap Extension Density, Gap Open Density can be computed in $O(mn)$ time.

Similar to Gap Extension Density, the ratio described above is generally decreasing in alignment accuracy, so for the feature value we use 1 minus the ratio described above.

3.2.6 Gap Compatibility

As in cladistics, we encode the gapping pattern in the columns of an alignment as a binary state: residue (`1`), or null character (`0`). For an alignment in this encoding we then collapse together adjacent columns that have the same gapping pattern. We evaluate this reduced set of columns for *compatibility* by checking whether a perfect phylogeny exists on them, using the so-called “four gametes test” on pairs of columns. More specifically, a pair of columns passes the four gametes test if at most three of the four possible patterns `00`, `01`, `10`, `11` occur in the rows of these two columns. A so-called perfect phylogeny exists, in which the binary gapping pattern in each column is explained by a single insertion or deletion event on an edge of the tree, if and only if all pairs of columns pass this test. (See Gusfield, 1997 pages 462-463, or Estabrook et al., 1975.) The Gap Compatibility feature measures the fraction of pairs of columns in the reduced binary data that pass this test, which is a rough measure of how tree-like the gapping pattern is in the alignment. Rather than determining whether a *complete* column pair passes the four-gametes test, we can instead measure the *fraction* of a column pair that pass this test (the largest subset of rows that pass the test divided by the total number of rows), averaged over all pairs of columns. We find that this second version of the feature works better in practice, most likely because it is a less strict measure of the evolutionary compatibility of the gaps.

For each pair of columns, we can compute the encoding of each row in constant time, so we can collect the counts for the four-gametes states in linear time in the number of sequences for a given column pair. Since we must examine all pairs of columns, the running time for Gap Compatibility is quadratic in the number of columns. Evaluating this feature takes $O(m^2n)$ time for an alignment with m sequences and n columns.

3.2.7 Substitution Compatibility

Similar to Gap Compatibility, we encode the substitution pattern in the columns of an alignment by a binary state: using a reduced amino acid alphabet of equivalency classes, residues in the most prevalent equivalency class in the column are mapped to `1`, and all others to `0`. This feature measures the fraction of encoded column pairs that pass the four-gametes test, which again is a rough measure of how tree-like the substitution pattern is in the alignment. Again we tested have options for using both whole-column and fractional-column measurements; we find that fractional-column measurements give better accuracy. We also considered the standard reduced amino-acid alphabets with 6, 10, 15, and 20 equivalency classes, and find the 15-class alphabet, gives the strongest correlation with accuracy.

Just like Gap Compatibility, evaluating Substitution Compatibility takes $O(m^2 n)$ time.

3.2.8 Amino Acid Identity

This feature is usually called simply "percent identity." In each induced pairwise alignment, we measure the fraction of substitutions in which the residues have the same amino-acid equivalency class, where we use the reduced alphabet with 10 classes. The feature averages this fraction over all induced pairwise alignments.

We can compute Amino Acid Identity for a whole alignment by determining the frequency of each amino-acid class in each column, and summing the number of pairs of each alphabet element in a column. Computing amino-acid identity in this way takes $O(mn + n|\Sigma|)$ time for amino-acid equivalency Σ. Assuming $|\Sigma|$ is constant, this is $O(mn)$ time.

3.2.9 Secondary Structure Identity

This feature is like Amino Acid Identity, except instead of the protein's amino-acid sequence, we use the secondary-structure sequence predicted for the protein by `PSIPRED` (Jones, 1999), which is a string over the 3-letter secondary structure alphabet. Similar to the approach described for Amino Acid Identity, we can compute Secondary Structure Identity in $O(mn)$ time (where the structural alphabet here is $\{\alpha, \beta, \gamma\}$, so $|\Sigma| = 3$).

We also consider a second version that uses the secondary structure confidences, where instead of counting identities, we calculate the probability that a pair i, j of residues has the same secondary structure, by

$$p_\alpha(i) p_\alpha(j) \; + \; p_\beta(i) p_\beta(j) \; + \; p_\gamma(i) p_\gamma(j).$$

In this version, we cannot use the prior running-time reduction trick, and must examine all pairs of rows in a column, which takes total time $O(m^2 n)$ for and alignment with m rows and n columns.

3.2.10 Average Substitution Score

This computes the total score of all substitutions in the alignment, using a `BLSM62` substitution-scoring matrix (Henikoff and Henikoff, 1992) that has been shifted and scaled so the amino acid similarity scores are in the range $[0, 1]$. We can normalize this total score by the number of substitutions in the alignment, or by upper bound U given earlier so the feature value is comparable between alignments of the same sequences. We find that normalizing by U provides a feature value that correlates better with true accuracy.

Similar to the running-time reduction of Amino Acid Identity, we can count the frequency of each amino acid in each column of an alignment, and sum the `BLSM62` score for each possible amino-acid substitution multiplied by the product of the frequency for the two amino acids. This reduces the running time to $O(mn+n|\Sigma|^2)$, which is faster than considering all pairs of rows when $|\Sigma| < n$ (otherwise we can use the naïve $O(m^2n)$ approach).

3.2.11 Core Column Density

For this feature, we first predict *core columns* as those that only contain residues (and not gap characters) and whose fraction of residues that have the same amino acid equivalency class, for the 10-class alphabet, is above a threshold. The feature then normalizes the count of predicted core columns by the total number of columns in the alignment. We considered the standard reduced alphabets with 6, 10, 15, and 20 equivalency classes, and use the 10-class alphabet, as it gave the strongest correlation with true accuracy. We also tested various thresholds and found that a value of 0.9 gave the best trend.

Using the same trick described earlier for Amino Acid Identity, a "core" label can be assigned to the column in linear time, therefore we can evaluate this naïve Core Column Density in $O(mn)$ time. We will later develop a more sophisticated method for predicting core columns in an alignment in Chapter 9.

3.2.12 Information Content

This feature measures the average entropy of the alignment (Hertz and Stormo, 1999), by summing over the columns the log of the ratio of the abundance of a specific amino acid in the column over the background distribution for that amino acid, normalized by the number of columns in the alignment.

Amino-acid frequencies can be calculated in linear time for each column, and background frequencies for each amino acid can also be found in one pass across the whole alignment. We then evaluate information content in each column by making one pass over the frequencies for each element in the alphabet. Computing Information Content for an input alignment in $O(mn + m|\Sigma|)$ time for alphabet Σ. Once again, if we assume the alphabet size is constant, this running time is $O(mn)$.

We considered the standard reduced alphabets with 6, 10, 15, and 20 equivalence classes, and used the 10-class alphabet, which gave the strongest correlation with true accuracy.

3.2.13 Results

Figure 3.1 shows the correlation of each of these features described above with true accuracy. We describe this set of benchmarks and testing procedures in full detail in Chapter 6. Briefly, we collected a total of 861 benchmark alignments from the `BENCH` suite of Edgar (2009), which consists of 759 benchmarks, supplemented by a selection of 102 benchmarks from the `PALI` suite of Balaji et al. (2001). For each of these benchmarks we used the `Opal` aligner to produce a new alignment of the sequences under its default parameter setting. For each of these computed alignments, we know the underlying correct alignment, so we can evaluate the true accuracy of the computed alignment. We also calculated each of the 12 feature values for each of these alignments. The figure shows the correlation of each of the features with true accuracy, where each of the 861 circles in each plot is one benchmark with its true accuracy on the horizontal axis and feature function value on the vertical. Notice that while all of the features generally have a positive trend with true accuracy, the ranges of the feature values differ substantially.

This comprises the set of features considered for constructing the `Facet` accuracy estimator. The next section describes the software implementing the `Facet` estimator.

3.3 Software

We implemented the `Facet` estimator using the `Java` programming language. The software can be used in one of three ways:

(1) ***Command line*** – For a given set of sequences, the user must first run `PSIPRED` to predict the secondary structure for each unaligned sequence. The scripts provided put these predictions in a format that is readable by `Facet`. `Facet` can then be invoked for each alignment in `FASTA` or `CLUSTAL` format, and the result is returned via standard out. Details of running `Facet` for alignments of the same sequences is shown in Figure 3.3. The `Facet` coefficients can also be changed via command line options (not shown in the figure), which override the default feature coefficients.

(2) ***Application Programming Interface*** – Within another `Java` application, a user can obtain the `Facet` score of an alignment by first creating a `FacetAlignment` object, which encapsulates the sequence alignment information as well as the secondary structure predictions. The user can then invoke `Facet` through a method call. An example of how to use `Facet` through the API is shown in Figure 3.2. The `Facet` coefficients can be changed in the API via a second argument to the method call (not shown in the figure).

(3) ***Within `Opal`*** – When creating an alignment using the `Opal` aligner,

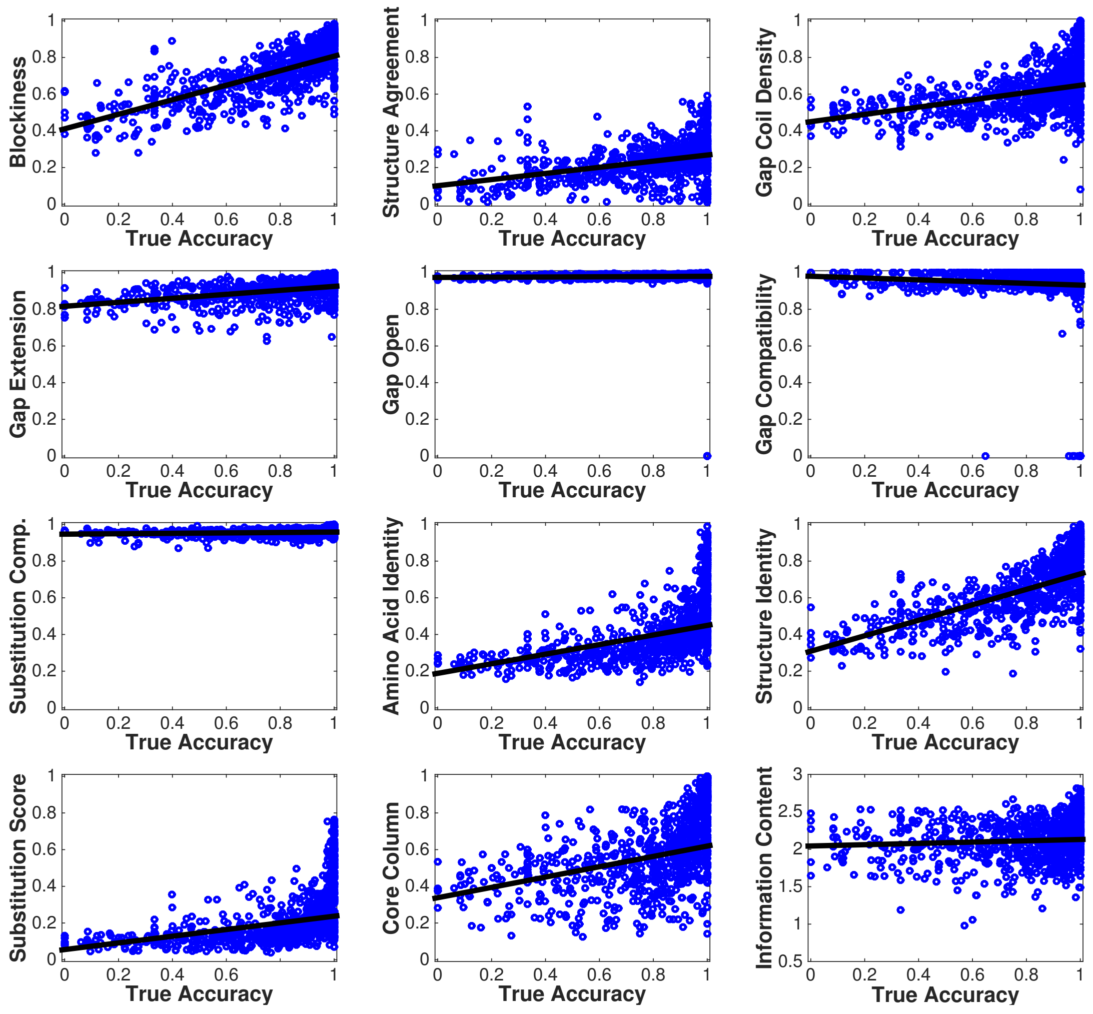
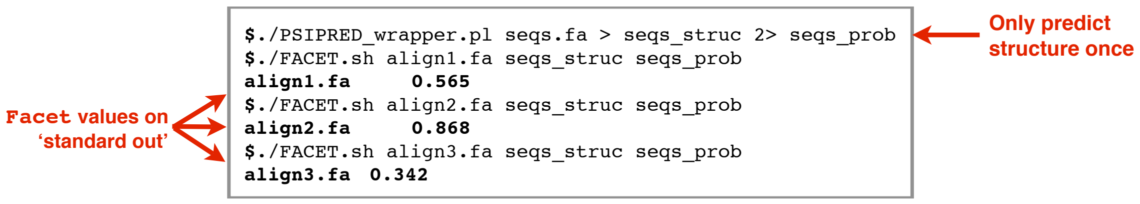


Figure 3.1: ***Correlation of features with true accuracy.*** The scatter plots show values of all twelve feature functions considered for the `Facet` estimator, on the 861 benchmarks used for testing (described in Chapter 6), using the default parameter setting for the `Opal` aligner. Each circle is one benchmark alignment plotted with its true accuracy on the horizontal axis (since we know the reference, we can calculate true accuracy) and its feature value on the vertical axis. The line shows is a weighted least-squares line where the weight for a benchmark is calculated to remove the bias towards benchmarks with high accuracy under the default parameter settings. The precise form of the weighting us described in detail in Chapter 6.

```
FacetAlignment align1 = new FacetAlignment(alignedSeqs1,strucPred,strucProb);
FacetAlignment align2 = new FacetAlignment(alignedSeqs2,strucPred,strucProb);

if(Facet.estimator(align1) > Facet.estimator(align2))
  return alignedSeqs1;
else
  return alignedSeqs2;
```

Figure 3.2: ***Using the `Facet` tool API.***



```
$./PSIPRED_wrapper.pl seqs.fa > seqs_struc 2> seqs_prob
$./FACET.sh align1.fa seqs_struc seqs_prob
align1.fa     0.565
$./FACET.sh align2.fa seqs_struc seqs_prob
align2.fa     0.868
$./FACET.sh align3.fa seqs_struc seqs_prob
align3.fa  0.342
```

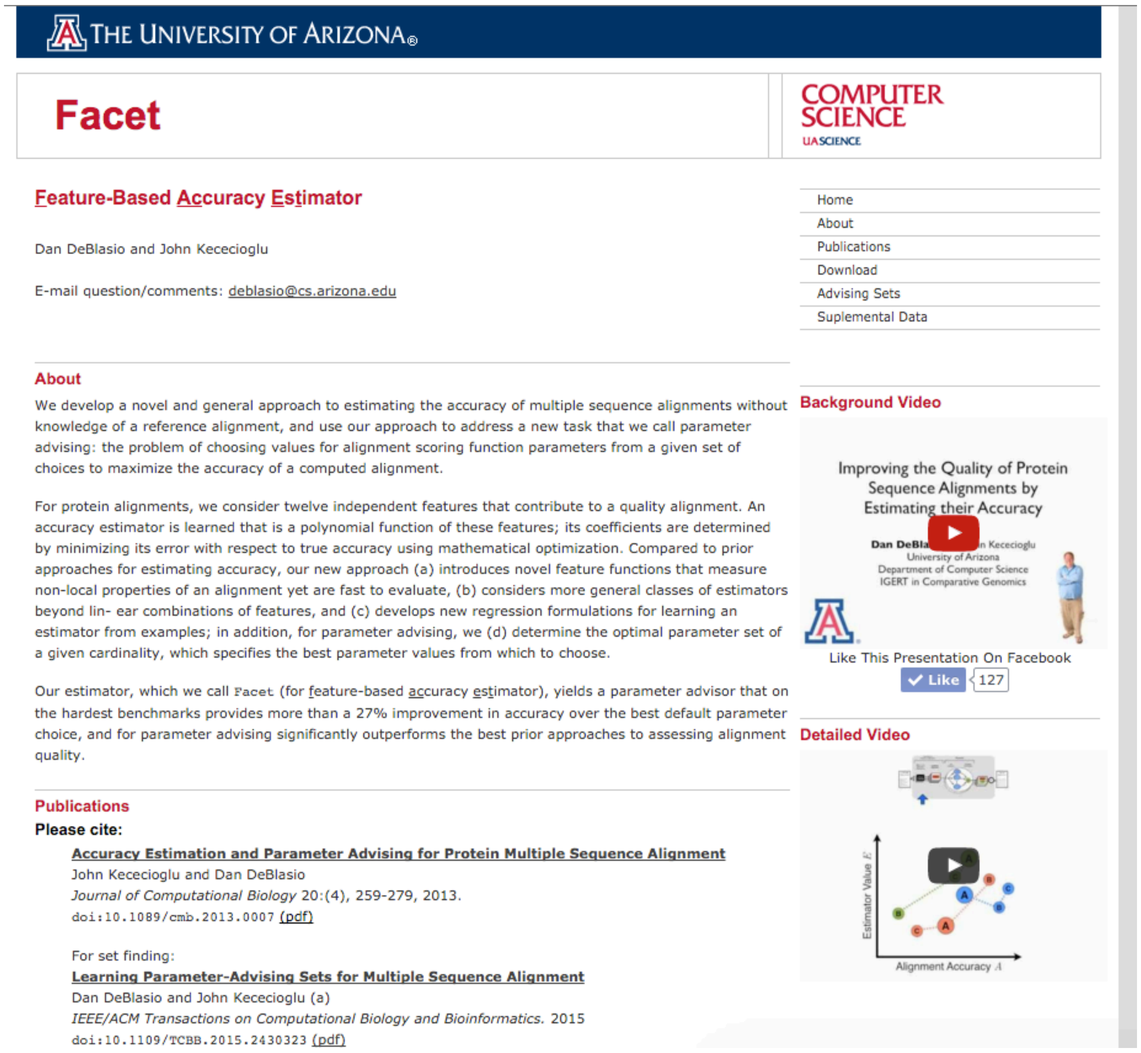
Figure 3.3: ***Using the `Facet` tool on the command line.***

a user can pass the alignment structure via the `--facet_structure` command-line option. The structure format given to `Facet` embedded within `Opal` is different from the format for stand-alone `Facet`. When the structure is given to `Opal`, the `Facet` score is printed to standard out, or can be included in the output filename. Section 6.6.1 gives details on the changes made to `Opal` related to `Facet` and parameter advising.

All three implementations of `Facet` can be found on the `Facet` website at `http://facet.cs.arizona.edu/` (see Figure 3.4). Along with `Facet`, and links to the `Opal` software, there are videos explaining our methodology, and supplementary data used in our experiments.

Summary

In this chapter, we have described several easily-computable feature functions for estimating alignment accuracy. Using these features in the framework described in Chapter 2 yields our new accuracy estimator `Facet`. We later give the coefficients for the feature functions, when trained on example alignments from benchmarks with known reference alignments, in Chapter 6.


THE UNIVERSITY OF ARIZONA®
COMPUTER SCIENCE
UA SCIENCE
Home
About
Publications
Download
Advising Sets
Suplemental Data
Improving the Quality of Protein Sequence Alignments by Estimating their Accuracy
University of Arizona
Department of Computer Science
IGERT in Comparative Genomics
Estimator Value E
Alignment Accuracy A

Facet

Feature-Based Accuracy Estimator

Dan DeBlasio and John Kececioglu

E-mail question/comments: deblasio@cs.arizona.edu

About

We develop a novel and general approach to estimating the accuracy of multiple sequence alignments without knowledge of a reference alignment, and use our approach to address a new task that we call parameter advising: the problem of choosing values for alignment scoring function parameters from a given set of choices to maximize the accuracy of a computed alignment.

For protein alignments, we consider twelve independent features that contribute to a quality alignment. An accuracy estimator is learned that is a polynomial function of these features; its coefficients are determined by minimizing its error with respect to true accuracy using mathematical optimization. Compared to prior approaches for estimating accuracy, our new approach (a) introduces novel feature functions that measure non-local properties of an alignment yet are fast to evaluate, (b) considers more general classes of estimators beyond lin- ear combinations of features, and (c) develops new regression formulations for learning an estimator from examples; in addition, for parameter advising, we (d) determine the optimal parameter set of a given cardinality, which specifies the best parameter values from which to choose.

Our estimator, which we call `Facet` (for feature-based accuracy estimator), yields a parameter advisor that on the hardest benchmarks provides more than a 27% improvement in accuracy over the best default parameter choice, and for parameter advising significantly outperforms the best prior approaches to assessing alignment quality.

Background Video

Like This Presentation On Facebook

Like 127

Detailed Video

Publications

Please cite:

Accuracy Estimation and Parameter Advising for Protein Multiple Sequence Alignment
John Kececioglu and Dan DeBlasio
Journal of Computational Biology 20:(4), 259-279, 2013.
doi:10.1089/cmb.2013.0007 (pdf)

For set finding:
Learning Parameter-Advising Sets for Multiple Sequence Alignment
Dan DeBlasio and John Kececioglu (a)
IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2015
doi:10.1109/TCBB.2015.2430323 (pdf)

Figure 3.4: ***The Facet Website.***

CHAPTER 4

The Optimal Advisor Problem

Overview

In this chapter, we define the problem of constructing an optimal advisor: finding both the estimator coefficients and advisor set that give the highest average advising accuracy. We can also restrict this problem to just finding optimal estimator coefficients for a given advisor set and finding an optimal advisor set for a given estimator. The optimal advisor problem is NP-complete (as are the restrictied optimal estimator and optimal advisor set problems).

This chapter was adapted from portions of previous publications (DeBlasio and Kececioglu, 2014a, 2016).

4.1 Introduction

A parameter advisor has two components: (i) the advisor estimator, which ranks alternate alignments that the advisor will choose among; and (ii) the advisor set, which should be small but still provide for each input at least one good alternate alignment that the advisor can choose. These two components are very much interdependent. A setting of estimator coefficients may work well for one advisor set, but may not be able to distinguish accurate alignments for another. Similarly for a given advisor set, one setting of advisor coefficients may work well while another may not.

An *optimal advisor* is both an advisor estimator and and advisor set that together produce the highest average advising accuracy on a collection of benchmarks. In this chapter, we consider the problem of constructing an optimal advisor. We also discuss restrictions of this problem to finding an optimal advisor set for a given estimator, or an optimal estimator for a given advisor set. All three versions of the problem are NP-complete.

4.2 Learning an optimal advisor

We now define the computational problem of learning an optimal advisor. The problem has several variations, depending on whether the advisor's estimator or set of parameter choices are fixed: (a) simultaneously finding both the best set and estimator, (b) finding the best *set* of parameter choices to use with a given estimator, and (c) finding the best *estimator* to use with a given set of parameter choices. We assume throughout that the features used by the advisor's estimator are given and fixed.

From a machine learning perspective, the problem formulations find an advisor that has optimal accuracy on a collection of training data. The underlying training data is

- a suite of *benchmarks*, where each benchmark B_i in the suite consists of a set of sequences to align, together with a *reference alignment* R_i for these sequences that represents their "correct" alignment, and
- a collection of *alternate alignments* of these benchmarks, where each alternate alignment A_{ij} results from aligning the sequences in benchmark i using a parameter choice j that is drawn from a given universe U of parameter choices.

Here a *parameter choice* is an assignment of values to all the parameters of an aligner that may be varied when computing an alignment. Typically an aligner has multiple parameters whose values can be specified, such as the substitution scoring matrix and gap penalties for its alignment scoring function. We represent a parameter choice by a vector whose components assign values to all these parameters. (So for protein sequence alignment, a typical parameter choice is a 3-vector specifying the (i) substitution matrix, (ii) gap-open penalty, and (iii) gap-extension penalty.) The universe U of parameter choices specifies all the possible parameter choices that might be used for advising. A particular advisor will use a subset $P \subseteq U$ of parameter choices that it considers when advising. In the special case $|P| = 1$, the single parameter choice in set P that is available to the advisor is effectively a *default* parameter choice for the aligner.

Note that since a reference alignment R_i is known for each benchmark B_i, the true accuracy of each alternate alignment A_{ij} for benchmark B_i can be measured by comparing alignment A_{ij} to the reference R_i. Thus for a set $P \subseteq U$ of parameter choices available to an advisor, the most accurate parameter choice $j \in P$ to use on benchmark B_i can be determined in principle by comparing the resulting alternate alignments A_{ij} to R_i and picking the one of highest true accuracy. When aligning sequences in practice, a reference alignment is not known, so an advisor will instead use its estimator to pick the parameter choice $j \in P$ whose resulting alignment A_{ij} has highest *estimated* accuracy.

In the problem formulations below, this underlying training data is summarized by

- the *accuracies* a_{ij} of the alternate alignments A_{ij}, where accuracy a_{ij} measures how well the computed alignment A_{ij} agrees with the reference alignment R_i, and
- the *feature vectors* F_{ij} of these alignments A_{ij}, where each vector F_{ij} lists the values for A_{ij} of the estimator's feature functions.

As we have defined in Chapter 2, for an estimator that uses t feature functions, each feature vector F_{ij} is a vector of t feature values,

$$F_{ij} \;=\; (g_{ij1}\; g_{ij2}\; \cdots\; g_{ijt}),$$

where each feature value g_{ijh} is a real number satisfying $0 \leq g_{ijh} \leq 1$. Feature vector F_{ij} is used by the advisor to evaluate its accuracy estimator E on alignment A_{ij}. Let the *coefficients* of the estimator E be given by vector

$$c \;=\; (c_1\; c_2\; \cdots\; c_t).$$

Then the value of accuracy estimator E on alignment A_{ij} is given by the inner product

$$E_c(A_{ij}) \;=\; c \cdot F_{ij} \;=\; \sum_{1 \le h \le t} c_h \, g_{ijh}. \tag{4.1}$$

Informally, the objective function that the problem formulations seek to maximize is the average accuracy achieved by the advisor across the suite of benchmarks in the training set. The benchmarks may be nonuniformly weighted in this average to correct for bias in the training data, such as the over-representation of easy benchmarks that typically occurs in standard benchmark suites.

A subtle issue that the formulations must take into account is that when an advisor is selecting a parameter choice via its estimator, there can be ties in the estimator value, so there may not be a unique parameter choice that maximizes the estimator. In this situation, we assume that the advisor *randomly* selects a parameter choice among those of maximum estimator value. Given this randomness, we measure the performance of an advisor on an input by its *expected* accuracy on that input.

Furthermore, in practice any accuracy estimator inherently has *error* (otherwise it would be equivalent to true accuracy), and a robust formulation for learning an advisor should be tolerant of error in the estimator. Let $\epsilon \ge 0$ be a given error *tolerance*, and P be the set of parameter choices used by an advisor. We define the set $\mathcal{O}_i(P)$ of parameter choices that the advisor could potentially *output* for benchmark B_i as

$$\mathcal{O}_i(P) \;=\; \Big\{ j \in P \;:\; E_c(A_{ij}) \;\ge\; e_i^* \,-\, \epsilon \Big\}, \tag{4.2}$$

where $e_i^* \;:=\; \max\big\{ E_c(A_{i\tilde{\jmath}}) \;:\; \tilde{\jmath} \in P \big\}$ is the maximum estimator value on benchmark B_i. The parameter choice output by an advisor on benchmark B_i is selected uniformly at random among those in $\mathcal{O}_i(P)$. Note that when $\epsilon = 0$, set $\mathcal{O}_i(P)$ is simply the set of parameter choices that are tied for maximizing the estimator. A nonzero tolerance $\epsilon > 0$ can aid in learning an advisor that has improved generalization to testing data.

The *expected accuracy* achieved by the advisor on benchmark B_i using set P is then

$$\mathcal{A}_i(P) \;=\; \frac{1}{|\mathcal{O}_i(P)|} \sum_{j \,\in\, \mathcal{O}_i(P)} a_{ij}\,. \tag{4.3}$$

In learning an advisor, we seek a set P that maximizes the advisor's expected accuracy $\mathcal{A}_i(P)$ on the training benchmarks B_i.

Formally, we want an advisor that maximizes the following *objective function*,

$$f_c(P) \;=\; \sum_i w_i \; \mathcal{A}_i(P)\,, \tag{4.4}$$

where i indexes the benchmarks, and w_i is the weight placed on benchmark B_i. (The benchmark weights are to correct for possible sampling bias in the training data.) In words, objective $f_c(P)$ is the expected accuracy of the parameter choices selected by the advisor averaged across the weighted training benchmarks, using advisor set P and the estimator given by coefficients c. We write the objective function as $f(P)$ without subscript c when the estimator coefficient vector c is fixed or understood from context.

We use the following *argmin* and *argmax* notation. For a function f and a subset S of its domain,

$$\operatorname{argmin}\bigl\{ f(x) \;:\; x \in S \bigr\}$$

denotes the set of *all* elements of S that achieve the minimum value of f, or in other words, the set of minimizers of f on S. Similarly, argmax is used to denote the set of maximizers.

4.2.1 Optimal Advisor

We first define the problem of finding an *optimal advisor*: that is, simultaneously finding an advisor estimator and an advisor set that together yields the highest average advising accuracy.

In the problem definition,

- n is the number of benchmarks, and
- t is the number of alignment features.

Set $\mathcal{Q}$ denotes the set of rational numbers.

Definition 1 (Optimal Advisor) The *Optimal Advisor* problem takes as input

- cardinality bound $k \geq 1$,
- universe U of parameter choices,
- weights $w_i \in \mathcal{Q}$ on the training benchmarks B_i, where each $w_i \geq 0$ and $\sum_i w_i = 1$,
- accuracies $a_{ij} \in \mathcal{Q}$ of the alternate alignments A_{ij}, where each $0 \leq a_{ij} \leq 1$,
- feature vectors $F_{ij} \in \mathcal{Q}^t$ for the alternate alignments A_{ij}, where each feature value g_{ijh} in vector F_{ij} satisfies $0 \leq g_{ijh} \leq 1$, and
- error tolerance $\epsilon \in \mathcal{Q}$ where $\epsilon \geq 0$.

The output is

- estimator coefficient vector $c \in \mathcal{Q}^t$, where each coefficient c_i in vector c satisfies $c_i \geq 0$ and $\sum_{1 \leq i \leq t} c_i = 1$, and
- set $P \subseteq U$ of parameter choices for the advisor, with $|P| \leq k$,

that maximizes objective $f_c(P)$ given by equation (4.4).

$\square$

4.2.2 Advisor Set

We can restrict the optimal advisor problem to finding an *optimal set* of parameter choices for advising with a given estimator.

Definition 2 (Advisor Set) The *Advisor Set* problem takes as input

- weights w_i on the benchmarks,
- accuracies a_{ij} of the alternate alignments,
- feature vectors F_{ij} for the alternate alignments,
- coefficients $c = (c_1 \cdots c_t) \in \mathcal{Q}^t$ for the estimator, where each $c_i \geq 0$ and $\sum_{1 \leq i \leq t} c_i = 1$, and

- error tolerance ϵ.

The output is

- advisor set P

that maximizes objective $f_c(P)$ given by equation (4.4). □

4.2.3 Advisor Estimator

Similarly, we can define the problem of finding an *optimal estimator* where the set of parameter choices for the advisor is now given.

Definition 3 (Advisor Estimator) The *Advisor Estimator* problem takes as input

- weights w_i on the benchmarks,
- accuracies a_{ij} of the alternate alignments,
- feature vectors F_{ij} for the alternate alignments,
- advisor set P, and
- error tolerance ϵ.

The output is

- coefficients $c = (c_1 \cdots c_t) \in \mathcal{Q}^t$ for the estimator, where each $c_i \geq 0$ and $\sum_{1 \leq i \leq t} c_i = 1$,

that maximize objective $f_c(P)$ given by equation (4.4). □

For Advisor Estimator, resolving ties to pick the worst among the parameter choices that maximize the estimator, as in the definition of $\mathcal{A}(i)$ in equation (4.4), is crucial, as otherwise the problem formulation becomes degenerate. If the advisor is free to pick any of the tied parameter choices, it can pick the tied one with highest true accuracy; if this is allowed, the optimal estimator c^* that is found by the formulation would degenerate to the flattest possible estimator that evaluates all parameter choices as equally good (since the degenerate flat estimator would make the advisor appear to match the performance of a perfect oracle on set P). Resolving ties in the worst-case way eliminates this degeneracy.

4.3 Complexity of learning optimal advisors

We now prove that Advisor Set, the problem of learning an optimal parameter set for an advisor (given by Definition 2 of Section 4.2) is NP-complete, and hence is unlikely to be efficiently solvable in the worst-case. As is standard, we prove NP-completeness for a decision version of this optimization problem, which is a version whose output is a yes/no answer (as opposed to a solution that optimizes an objective function).

The *decision version* of Advisor Set has an additional input $\ell \in \mathcal{Q}$, which will lower bound the objective function. The decision problem is to determine, for the input instance $k, U, w_i, a_{ij}, F_{ij}, c, \epsilon, \ell$, whether or not there exists a set $P \subseteq U$ with $|P| \le k$ for which the objective function has value $f_c(P) \ge \ell$.

Theorem 3 (NP-completeness of Advisor Set) *The decision version of Advisor Set is NP-complete.*

Proof We use a reduction from the *Dominating Set* problem, which is NP-complete (Garey and Johnson, 1979, problem GT2). The input to Dominating Set is an undirected graph $G = (V, E)$ and an integer k, and the problem is to decide whether or not G contains a vertex subset $S \subseteq V$ with $|S| \le k$ such that every vertex in V is in S or is adjacent to a vertex in S. Such a set S is called a *dominating set* for G.

Given an instance G, k of Dominating Set, we construct an instance $U, w_i, a_{ij}, F_{ij}, c, \epsilon, \ell$ of the decision version of Advisor Set as follows. For the cardinality bound use the same value k, for the number of benchmarks take $n = |V|$, and index the universe of parameter choices by $U = \{1, \ldots, n\}$; have only one feature ($d = 1$) with estimator coefficients $c = 1$; use weights $w_i = 1/n$, error tolerance $\epsilon = 0$, and lower bound $\ell = 1$. Let the vertices of G be indexed

$V = \{1, \ldots, n\}$. (So both the set of benchmarks and the universe of parameter choices in essence correspond to the set of vertices V of graph G.) Define the *neighborhood* of vertex i in G to be $N(i) := \{j : (i,j) \in E\} \cup \{i\}$, which is the set of vertices adjacent to i, including i itself. For the alternate alignment accuracies, take $a_{ij} = 1$ when $j \in N(i)$; otherwise, $a_{ij} = 0$. For the feature vectors, assign $F_{ij} = a_{ij}$.

We claim G, k is a yes-instance of Dominating Set iff $k, U, w_i, a_{ij}, F_{ij}, c, \epsilon, \ell$ is a yes-instance of Advisor Set.

To show the forward implication, suppose G has a dominating set $S \subseteq V$ with $|S| \leq k$, and consider the advisor set $P = S$. With the above construction, for every benchmark, set $\mathcal{O}_i(P) = N(i) \cap S$, which is nonempty (since S is a dominating set for G). So $\mathcal{A}_i(P) = 1$ for all benchmarks. Thus for this advisor set P, the objective function has value $f_c(P) = 1 \geq \ell$.

For the reverse implication, suppose advisor set P achieves objective value $\ell = 1$. Since P achieves value 1, for every benchmark it must be that $\mathcal{A}_i(P) = 1$. By construction of the a_{ij}, this implies that in G every vertex $i \in V$ is in P or is adjacent to a vertex in P. Thus set $S = P$, which satisfies $|S| \leq k$, is a dominating set for G.

This reduction shows Advisor Set is NP-hard, as the instance of Advisor Set can be constructed in polynomial time. Furthermore, it is in NP, as we can nondeterministically guess an advisor set P, and then check whether its cardinality is at most k and its objective value is at least ℓ in polynomial time. Thus Advisor Set is NP-complete. $\square$

Note that the proof of Theorem 3 shows Advisor Set is NP-complete for the special case of a *single feature*, error tolerance zero, when all accuracies and feature values are binary, and benchmarks are uniformly weighted.

In general, we would like to find an optimal *parameter advisor*, which requires simultaneously finding both the best possible parameter *set* and the best possible accuracy *estimator*. We define the general problem of constructing an optimal parameter advisor as follows.

The *decision version* of Optimal Advisor, similar to the decision version of Advisor Set, has an additional input ℓ that lower bounds the objective function.

We next prove that Optimal Advisor is NP-complete. While its NP-hardness follows from Advisor Set, the difficulty is in proving that this more general problem is still in the class NP.

Theorem 4 (NP-completeness of Optimal Advisor) *The decision version of Optimal Advisor is NP-complete.*

Proof The proof of Theorem 3 shows Advisor Set remains NP-hard for the special case of a single feature. To prove the decision version of Optimal Advisor is NP-hard, we use *restriction*: we simply reduce Advisor Set with a single feature to Optimal Advisor (reusing the instance of Advisor Set for Optimal Advisor). On this restricted input with $d = 1$, Optimal Advisor is equivalent to Advisor Set, so Optimal Advisor is also NP-hard.

We now show the general Optimal Advisor problem is in class NP. To decide whether its input is a yes-instance, after first nondeterministically guessing parameter set $P \subseteq U$ with $|P| \leq k$, we then make for each benchmark i a nondeterministic guess for its sets $\mathcal{O}_i(P)$ and $\mathcal{M}_i(P) := \operatorname{argmax}\big\{c \cdot F_{ij} \,:\, j \in P\big\}$, *without* yet knowing the coefficient vector c. Call $\widetilde{O}_i$ the guess for set $\mathcal{O}_i(P)$, and $\widetilde{M}_i$ the guess for set $\mathcal{M}_i(P)$, where $\widetilde{M}_i \subseteq \widetilde{O}_i \subseteq P$. To check whether a coefficient vector c exists that satisfies $\mathcal{O}_i(P) = \widetilde{O}_i$ and $\mathcal{M}_i(P) = \widetilde{M}_i$, we construct the following linear program with variables $c = (c_1 \cdots c_d)$ and ξ. The objective function for the linear program is to maximize the value of variable ξ. The constraints are: $c_h \geq 0$ and $\sum_{1 \leq h \leq d} c_h = 1$; $0 \leq \xi \leq 1$; for all benchmarks i and all parameter choices $j^* \in \widetilde{M}_i$ and $j \notin \widetilde{M}_i$,

$$c \cdot F_{ij^*} \;\geq\; c \cdot F_{ij} \;+\; \xi\,;$$

for all benchmarks i and all parameter choices $j, \tilde{\jmath} \in \widetilde{M}_i$,

$$c \cdot F_{ij} \;=\; c \cdot F_{i\tilde{\jmath}}\,;$$

for all benchmarks and all parameter choices $j^* \in \widetilde{M}_i$ and $j \in \widetilde{O}_i$,

$$c \cdot F_{ij} \;\geq\; c \cdot F_{ij^*} \;-\; \epsilon\,.$$

This linear program can be solved in polynomial time. If it has a feasible solution, then it has an optimal solution (as its objective function is bounded). In an optimal solution c^*, ξ^* we check whether $\xi^* > 0$. If this condition holds, the guessed sets $\widetilde{O}_i$, $\widetilde{M}_i$, correspond to actual sets $\mathcal{O}_i(P)$ and $\mathcal{M}_i(P)$ for an estimator. For each benchmark i, we then evaluate $\mathcal{A}_i(P)$, and check whether $\sum_i w_i\, \mathcal{A}_i(P) \,\geq\, \ell$. Note that after guessing the sets P, $\widetilde{O}_i$, and $\widetilde{M}_i$, the rest of the computation runs in polynomial time. Thus Optimal Advisor is in NP. $\square$

Theorem 5 (NP-completeness of Advisor Estimator) *The decision version of Advisor Estimator is NP-complete.*

Proof To show Advisor Estimator is NP-hard, we use a similar reduction from Dominating Set that we used in proving Theorem 3. Given an instance G, k of Dominating Set, we construct an instance $w_i, a_{ij}, F_{ij}, P, \epsilon, \ell, \delta$ of the decision version of Advisor Estimator, where we use the same cardinality bound k, number of benchmarks and parameter choices $n = |V|$, weights $w_i = 1/n$, error tolerance $\epsilon = 0$, accuracies a_{ij} again defined as before, and lower bound $\ell = 1$, as we did for Advisor Set. For the set P of parameter choices for the advisor, we take $P = \{1, \ldots, n\}$. The number of features is now $t = n$. (So in essence the set of benchmarks, the advisor set, and the set of features all coincide with the set of vertices V.) For the feature vectors we take $F_{ij} = (0 \cdots 0\ a_{ij}\ 0 \cdots 0)$ which has value a_{ij} at location j. This is equivalent to a feature vector F_{ij} that is all zeroes, except for a 1 at location j if $j = i$ or vertex j is adjacent to vertex i in G. For the precision lower bound we take $\delta = 1/k$. Note that this instance of Advisor Estimator can be constructed in polynomial time.

We claim that G, k is a yes-instance of Dominating Set iff $w_i, a_{ij}, F_{ij}, P, \epsilon, \ell, \delta$ is a yes-instance of Advisor Estimator. To show the reverse implication, first notice that with the chosen δ, coefficient vector c can have at most k nonzero coefficients (since if c has more than k nonzero coefficients, $\sum_i c_i > k\,\delta = 1$, a contradiction). Let feature subset $S \subseteq V$ be all indices i at which $c_i > 0$. We call S the *support* of c, and by our prior observation $|S| \leq k$. By construction of the feature vectors, $c \cdot F_{ij} = c_j$ if $j \in N(i)$; otherwise, $c \cdot F_{ij} = 0$. This further implies that $\mathcal{A}_i(P) = 1$ if $S \cap N(i)$ is nonempty; otherwise, $\mathcal{A}_i(P) = 0$. So if there exists coefficient vector c such that the objective function achieves value 1, then the support S of c gives a vertex subset $S \subseteq V$ that is a dominating set for G. For the forward implication, given a dominating set $S \subseteq V$ for G, take for the estimator coefficients $c_i = 1/|S|$ if $i \in S$, and $c_i = 0$ otherwise. The nonzero coefficients of this vector c have value at least δ, and by the same reasoning as above, each $\mathcal{A}_i(P) = 1$ as S is a dominating set, so the estimator given by this vector c yields an advisor that achieves objective value 1, which proves the claim.

We can show Advisor Estimator is in class NP using the same construction used for proving Optimal Advisor is in class NP. For each benchmark we can make a nondeterministic choice for its set $\widetilde{\mathcal{O}}_i(P)$, and compute $\widetilde{\mathcal{M}}_i$. We can then construct a linear program to determine if these guesses are actual sets for the estimator. The guesses and solution of the linear program can be performed in polynomial time. Thus Advisor Estimator is in NP. $\square$

Summary

In this chapter, we have formally defined the problem of finding an optimal advisor and two related problems of finding an optimal advisor set and an optimal advisor estimator. We then proved that all three problems (Optimal Advisor, Advisor Set, and Advisor Estimator) are NP-complete. In the next chapter, we describe practical approaches to the Advisor Set problem, and how to model all three problems by mixed-integer linear programming (MILP).

CHAPTER 5

Constructing Advisor

Overview

In this chapter, we consider the problem of learning an opimal set of parameter choices for a parameter advisor. We consider two forms of the advisor sets problem: (i) sets that are estimator-unaware (and are optimal for a prefect estimator called an oracle), and (ii) sets that are optimal for a given accuracy estimator. In this context the optimal advisor set is one that maximizes the average true accuracy of the resulting parameter advisor, over a collection of training benchmarks. Chapter 4, we proved in that learning an optimal set for an advisor is NP-complete. Here we can model the problem of finding optimal advisor sets as an integer linear program (ILP). We find this ILP cannot be solved to optimality in practice, so we go on to develop an efficient approximation algorithm for this problem that finds near-optimal sets, and prove a tight bound on its approximation ratio.

This chapter was adapted from portions of previous publications (DeBlasio and Kececioglu, 2014a, 2015).

5.1 Introduction

In Chapter 4, we introduced the *advisor set* problem and showed that it is NP-complete. In this chapter, we show how to model the problem of finding optimal advisor sets, and more generally finding optimal advisor, using integer linear programming. We have found that in practice these integer linear programming models are not solvable to optimality even on very small inputs. Consequently, Section 5.3 develops an efficient *approximation algorithm*, that is guaranteed to find near-optimal advisor sets for a given estimator.

In this chapter we consider how to learn sets of parameter choices for a *realistic advisor*, where these sets are tailored to the actual estimator used by the advisor (as opposed to finding parameter sets for a perfect but unattainable oracle advisor). While learning such sets that are oprimal is NP-complete, there is an efficient greedy approximation algorithm for this learning problem, and we derive a tight bound on its worst-case approximation ratio. Experiments show that the *greedy* parameter sets found by this approximation algorithm, using `Facet`, `TCS`, `MOS`, `PredSP`, or `GUIDANCE` as the advisor's accuracy estimator, outperform optimal *oracle* sets at all cardinalities. Furthermore, on the training data, for some estimators these suboptimal greedy sets perform surprisingly close to optimal *exact* sets found by exhaustive search. Moreover, these greedy sets actually *generalize* better than exact sets. As a consequence, on testing data, for some estimators the greedy sets output by the approximation algorithm can actually give superior performance to exact sets for parameter advising.

5.2 Constructing optimal advisors by integer linear programming

We now show how to construct optimal advisors by integer linear programming. Recall that an *integer linear program* (ILP) is an optimization problem with a collection of integer-valued variables, an objective function to optimize that is linear in these variables, and constraints that are linear inequalities in the variables. Our formulations of Advisor Coefficients and Optimal Advisor are actually so-called *mixed-integer* programs, where some of the variables are real-valued, while Advisor Set has all integer variables.

The integer linear programming formulations we give below actually model a more general version of the advising problems. The advising problems in Section 4.2 define the advisor $\mathcal{A}$ so that it carefully resolves ties among the parameter choices that achieve the optimum value of the estimator, by picking from this tied set the parameter choice that has lowest true accuracy. (This finds a solution that has the best possible average accuracy, even in the worst case.) We extend the definition of advisor $\mathcal{A}$ to now pick from a larger set of *near-optimal* parameter choices with respect to the estimator. To make this precise, for benchmark i, set P of parameter choices, and a real-value $\delta \geq 0$, let

$$M_\delta(i) \;:=\; \Big\{ j \in P \;:\; c \cdot F_{ij} \;\geq\; \max_{k \in P} \big\{ c \cdot F_{ik} \big\} \;-\; \delta \Big\}.$$

Set $M_\delta(i)$ is the near-optimal parameter choices that are within δ of maximizing the estimator for benchmark i. (So $M_\delta(i) \supseteq \mathrm{argmax}_{j \in P} \{c \cdot F_{ij}\}$, with equality when $\delta = 0$.) We then extend the definition of the advisor $\mathcal{A}$ in equation (4.4) for $\delta \geq 0$ to

$$\mathcal{A}(i) \;\in\; \mathrm{argmin} \Big\{ a_{ij} \;:\; j \;\in\; M_\delta(i) \Big\}. \tag{5.1}$$

At $\delta = 0$, this coincides with the original problem definitions. The extension to $\delta > 0$ is designed to boost the *generalization* of optimal solutions (in other words, to find a solution that is not over fit to the training data) when we do cross-validation experiments on independent training and test sets as in Chapter 6. We give integer linear programming formulations for this extended definition of our advising problems.

5.2.1 Modeling the Advisor Set Problem

The integer linear program (ILP) for Advisor Set has three classes of *variables*, which all take on binary values $\{0, 1\}$. Variables x_{ij}, for all benchmarks i and all parameter choices j from the universe, encode the advisor $\mathcal{A}$: $x_{ij} = 1$ if the advisor uses choice j on benchmark i; otherwise, $x_{ij} = 0$. Variables y_j, for all parameter choices j from the universe, encode the set P that is found by Advisor Set: $y_j = 1$ iff $j \in P$. Variables z_{ij}, for all benchmarks i and parameter choices j, encode the parameter choice in P with highest estimator value for benchmark i: if $z_{ij} = 1$ then $j \in \mathrm{argmax}_{k \in P}\; c \cdot F_{ik}$. This argmax set may contain several choices j, and in this situation the ILP given below arbitrarily selects one such choice j for which $z_{ij} = 1$.

For convenience, the description of the ILP below also refers to the new *constants* e_{ij}, which are the estimator values of the alternate alignments A_{ij}: for the fixed estimator c for Advisor Set, $e_{ij} = c \cdot F_{ij}$.

The *objective function* for the ILP is to maximize

$$\sum_i w_i \sum_j a_{ij}\, x_{ij}. \tag{5.2}$$

In this function, the inner sum $\sum_j a_{ij}\, x_{ij}$ will be equal to $a_{i,\mathcal{A}(i)}$, as the x_{ij} will capture the (unique) parameter choice that advisor $\mathcal{A}$ makes for benchmark i. This objective is linear in the variables x_{ij}.

The *constraints* for the ILP fall into three classes. The *first class* ensures that variables y_j encode set P, and variables x_{ij} encode an assignment to benchmarks from P. The ILP has constraints

$$\sum_j y_j \;\leq\; k, \tag{5.3}$$

$$\sum_j x_{ij} \;=\; 1, \tag{5.4}$$

$$x_{ij} \;\leq\; y_j, \tag{5.5}$$

where equation (5.4) occurs for all benchmarks i, and inequality (5.5) occurs for all benchmarks i and all parameter choices j.

In the above, inequality (5.3) enforces $|P| \leq k$. Equations (5.4) force the advisor to select one parameter choice for every benchmark. Inequalities (5.5) enforce that the advisor's selections must be parameter choices that are available in P.

The *second class* of constraints ensure that variables z_{ij} encode a parameter choice from P with highest estimator value. To enforce that the z_{ij} encode an assignment to benchmarks from P,

$$\sum_j z_{ij} \;=\; 1, \tag{5.6}$$

$$z_{ij} \;\leq\; y_j, \tag{5.7}$$

where equation (5.6) occurs for all i, and inequality (5.7) occurs for all i and j. (In general, the z_{ij} will differ from the x_{ij}, as the advisor does not necessarily select the parameter choice with highest estimator value.) For all benchmarks i, and all parameter choices j and k from the universe with $e_{ik} < e_{ij}$, we have the inequality

$$z_{ik} + y_j \;\leq\; 1. \tag{5.8}$$

Inequalities (5.8) ensure that if a parameter choice k is identified as having the highest estimator value for benchmark i by $z_{ik} = 1$, there must not be any other parameter choice j in P that has higher estimator value on i. Note that the constants e_{ij} are known in advance, so inequalities (5.8) can be enumerated by sorting all j by their estimator value e_{ij}, and collecting the ordered pairs (k, j) from this sorted list.

The *third class* of constraints ensure that the parameter choices x_{ij} selected by the advisor correspond to the definition in equation (5.1): namely, among the parameter choices in P that are within δ of the highest estimator value from P for benchmark i, the parameter choice of lowest accuracy is selected. For all benchmarks i, all parameter choices j, and all parameters choices k and h with both $e_{ik}, e_{ih} \in [e_{ij} - \delta, e_{ij}]$ and $a_{ih} < a_{ik}$, we have the inequality

$$x_{ik} + y_h + z_{ij} \leq 2. \tag{5.9}$$

Inequalities (5.9) ensure that for the parameter choices that are within δ of the highest estimator value for benchmark i, the advisor only selects parameter choice k for i if k is within δ of the highest and there is no other parameter choice available in P within δ of the highest that has lower accuracy. Finally, for all benchmarks i and all parameter choices j and k with $e_{ik} < e_{ij} - \delta$, we have the inequality

$$x_{ik} + y_j \leq 1. \tag{5.10}$$

Inequalities (5.10) enforce that the advisor cannot select parameter choice k for i if the estimator value for k is below δ of an available parameter choice in P. (Inequalities (5.9) capture the requirements on parameter choices that are within δ of the highest, while inequalities (5.10) capture the requirements on parameter choices that are below δ of the highest.)

A truly remarkable aspect of this formulation is that the ILP is able to capture all the subtle conditions the advisor must satisfy through its static set of inequalities (listed at "compile time"), *without* knowing when the ILP is written what the optimal *set* P is, and hence without knowing what parameter choices in P have the highest estimator value for each benchmark.

To summarize, the ILP for Advisor Set has binary variables x_{ij}, y_j, and z_{ij}, and inequalities (5.3) through (5.10). For n benchmarks and a universe of m parameter choices, this is $O(mn)$ variables, and $O(m^2 n + m\widetilde{m}^2 n)$ constraints, where $\widetilde{m}$ is the maximum number of parameter choices that are within δ in estimator value of any given parameter choice. For small δ, typically $\widetilde{m} \ll m$, which leads to $O(m^2 n)$ constraints in practice. In the worst-case, though, the ILP has $\Theta(m^3 n)$ constraints.

We also have an alternate ILP formulation that adds $O(n)$ real-valued variables to capture the highest estimator value from P for each benchmark i, and only has $O(m^2 n)$ total constraints (so fewer constraints than the above ILP in the worst case), but its objective function is more involved, and attempts to solve the alternate ILP suffered from numerical issues.

5.2.2 Finding optimal Oracle Sets

While we would like to find advisor sets that are optimal for the actual accuracy estimator used by an advisor, in practice finding such optimal set seems very hard. We can, however, in practice find optimal advisor sets that are *estimator oblivious*, in the sense that the set-finding algorithm is unaware of the mistakes made by the advisor due to using an accuracy estimator rather than knowing true accuracy. More precisely, we can find an optimal advisor set for an advisor whose "estimator" is the true accuracy of an alignment. As mentioned previously, we call such an advisor an *oracle*.

To find an optimal oracle set, we use the same objective function described in equation 5.2, and equations 5.3-5.5 to make sure an alignment is only selected if the parameter that is used to generate it is chosen. Solving the ILP with only these constraints will yield an optimal advisor set for the oracle advisor. Note that ties in accuracy do not need to be resolved, as any alignment with a tied "estimator" value also has a tied accuracy value, and thus would not effect the objective value.

With the reduced number of variables and constraints in this modified ILPm we are able to find optimal oracle sets in practice even for large set cardinalities.

5.2.3 Modeling the Advisor Estimator Problem

We now describe how to modify the above ILP for Advisor Set to obtain an ILP for Advisor Coefficients. The modifications must address two issues: (a) the set P is now fixed; and (b) the estimator c is no longer fixed, so the enumeration of inequalities cannot exploit concrete estimator values. We can easily handle that set P is now part of the input by approproriately fixing the variables y_j with new equations: for all $j \in P$ add equation $y_j = 1$, and for all $j \notin P$ add $y_j = 0$.

To find the optimal estimator c, we add ℓ new real-valued variables $c_1, \ldots, c_\ell$ with the constraints $\sum_h c_h = 1$ and $c_h \geq 0$. We also add two new classes of binary-valued integer variables: (a) variable s_{ij}, for all benchmarks i and all parameter choices j, which has value 1 when the estimator value of parameter choice j on benchmark i, namely $c \cdot F_{ij}$, is within δ of the highest estimator value for i; and (b) variable t_{ijk}, for all benchmarks i and all parameter choices j and k, which has value 1 when $c \cdot F_{ij} > c \cdot F_{ik} - \delta$.

To set the values of the binary variables t_{ijk}, for all i, j, k we add the inequalities

$$t_{ijk} \;\; \geq \;\; c \cdot F_{ij} \; - \; c \cdot F_{ik} \; + \; \delta. \tag{5.11}$$

This inequality is linear in the variables $c_1, \ldots, c_\ell$. Note that the value of the estimator $c \cdot F_{ij}$ will always be in the range $[0, 1]$, and we are assuming that the constant $\delta \ll 1$. To set the values of the binary variables s_{ij}, for all i, j, k we add the inequalities

$$s_{ij} \;\; \geq \;\; t_{ijk} \; + \; t_{ikj} \; + \; z_{ik} \; - \; 2. \tag{5.12}$$

While the ILP only has to capture relationships between parameter choices that are in set P, we do not constrain the variables s_{ij} and t_{ijk} for parameter choices outside P to be 0, but allow the ILP to set them to 1 if needed for a feasible solution.

We now use the variables s_{ij} and t_{ijk} to express the relationships in the former inequalities (5.8) through (5.10). We replace inequality (5.8) by the following inequality over all i, j, k,

$$z_{ij} \; + \; y_k \;\; \leq \;\; 2 \; - \; \big(c \cdot F_{ik} \; - \; c \cdot F_{ij}\big). \tag{5.13}$$

We replace inequality (5.9) by the following inequality over all i, j, k with $a_{ik} < a_{ij}$,

$$x_{ij} \,+\, y_k \,+\, s_{ij} \,+\, s_{ik} \;\leq\; 3. \tag{5.14}$$

Finally, we replace inequality (5.10) by the following inequality over all i, j, k,

$$x_{ij} \,+\, y_k \;\leq\; 2 \,-\, \big(c \cdot F_{ik} \,-\, c \cdot F_{ij} \,-\, \delta\big). \tag{5.15}$$

To summarize, the ILP for Advisor Coefficients has binary variables x_{ij}, y_j, z_{ij}, s_{ij}, t_{ijk}, real variables c_h, constraints (5.6)–(5.7) and (5.11)–(5.15), plus the elementary constraints on the y_j and c_h. This is $O(m^2 n)$ variables and $O(m^2 n)$ constraints. While in general this is an enormous mixed-integer linear program, we are able to solve it to optimality for small, fixed sets P. Its difficulty increases with the size of P, and instances up to $|P| \leq 4$ can be solved in two days of computation.

5.2.4 Modeling the Optimal Advisor Problem

The ILP for Optimal Advisor is simply the above ILP for Advisor Coefficients where set P coincides with the entire universe of parameter choices: $P = \{1, \ldots, m\}$. Solving this ILP is currently beyond reach.

While very large integer linear programs can be solved to optimality in practice using modern solvers such as `CPLEX` (`IBM` Corporation, 2015) there is no known algorithm for integer linear programming that is efficient in the worst-case. Thus our reductions of the optimal advising problems to integer linear programming do not yield algorithms for these problems that are guaranteed to be efficient. On the other hand, Section 4.3 shows that our optimal advising problems are all NP-complete, so it is unlikely that *any* worst-case efficient algorithm for them exists.

5.3 Approximation algorithm for learning advisor sets

As Advisor Set is NP-complete, it is unlikely we can efficiently find advisor sets that are *optimal*; we can, however, efficiently find advisor sets that are guaranteed to be *close* to optimal, in the following sense. An α-*approximation algorithm* for a maximization problem, where $\alpha < 1$, is a polynomial-time algorithm that finds a feasible solution whose value under the objective function is at least factor α times the value of an optimal solution. Factor α is called the *approximation ratio*. In this section we show that for any constant ℓ with $\ell \leq k$, there is a simple approximation algorithm for Advisor Set that achieves approximation ratio ℓ/k.

For constant ℓ, the optimal advisor set of cardinality at most ℓ can be found in polynomial time by exhaustive search (since when ℓ is a constant there are polynomially-many subsets of size at most ℓ). The following natural approach to Advisor Set builds on this idea, by starting with an optimal advisor set of size at most ℓ, and greedily augmenting it to one of size at most k. Since augmenting an advisor set by adding a parameter choice can worsen its value under the objective function, even if augmented in the best possible way, the procedure `Greedy` given below outputs the best advisor set found across all cardinalities.

```
procedure Greedy($\ell, k$) begin
    Find an optimal subset $P \subseteq U$ of size $|P| \leq \ell$ that maximizes $f(P)$.
    $(\widetilde{P}, \widetilde{\ell}) := (P, |P|)$
    for cardinalities $\widetilde{\ell}+1, \ldots, k$ do begin
        Find parameter choice $j^* \in U - \widetilde{P}$ that maximizes $f(\widetilde{P} \cup \{j^*\})$.
        $\widetilde{P} := \widetilde{P} \cup \{j^*\}$
        if $f(\widetilde{P}) > f(P)$ then $P := \widetilde{P}$
    end
    output $P$
end
```

We now show this natural greedy procedure is an approximation algorithm for Advisor Set.

Theorem 6 (Approximation Ratio) *Procedure* `Greedy` *is an* (ℓ/k)*-approximation algorithm for Advisor Set with cardinality bound* k*, and any constant* ℓ *with* $\ell \leq k$*.*

Proof The basic idea of the proof is to use averaging over all subsets of size ℓ from the optimal advisor set of size at most k, in order to relate the objective function value of the set found by `Greedy` to the optimal solution.

To prove the approximation ratio, let

- P^* be the optimal advisor set of size at most k,
- $\widetilde{P}$ be the optimal advisor set of size at most ℓ,
- P be the advisor set output by `Greedy`,
- $\mathcal{S}$ be the set of all subsets of P^* that have size ℓ,
- $\widetilde{k}$ be the size of P^*, and
- $\widetilde{\ell}$ be the size of $\widetilde{P}$.

Note that if $\widetilde{k} < \ell$, then the greedy advisor set P is actually optimal and the approximation ratio holds. So assume $\widetilde{k} \geq \ell$, in which case $\mathcal{S}$ is nonempty. Then

$$
\begin{aligned}
f(P) \;&\geq\; f(\widetilde{P}) \\
&\geq\; \max_{Q \in \mathcal{S}} f(Q) \qquad (5.16) \\
&\geq\; \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} f(Q) \\
&=\; \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} \sum_{i} w_i \, \mathcal{A}_i(Q) \\
&=\; \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} \sum_{i} \sum_{j \in \mathcal{O}_i(Q)} \frac{w_i \, a_{ij}}{\bigl|\mathcal{O}_i(Q)\bigr|} \\
&=\; \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} \sum_{j \in Q} \sum_{i \,:\, j \in \mathcal{O}_i(Q)} \frac{w_i \, a_{ij}}{\bigl|\mathcal{O}_i(Q)\bigr|}\,, \qquad (5.17)
\end{aligned}
$$

where inequality (5.16) holds because $\widetilde{P}$ is an optimal set of size at most ℓ and each Q is a set of size ℓ, while equation (5.17) just changes the order of summation on i and j.

Note that for any subset $Q \subseteq P^*$ and any fixed parameter choice $j \in Q$, the following relationship on sets of benchmarks holds:

$$
\Bigl\{ i \;:\; j \in \mathcal{O}_i(P^*) \Bigr\} \;\subseteq\; \Bigl\{ i \;:\; j \in \mathcal{O}_i(Q) \Bigr\}, \qquad (5.18)
$$

since if choice j is within tolerance ϵ of the highest estimator value for P^*, then j is within ϵ of the highest value for Q.

Continuing from equation (5.17), applying relationship (5.18) to index i of the innermost sum and observing that the terms lost are nonnegative, yields the following inequality (5.19):

$$\begin{aligned} f(P) &\geq \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} \sum_{j \in Q} \sum_{i \,:\, j \in \mathcal{O}_i(Q)} \frac{w_i \, a_{ij}}{\left|\mathcal{O}_i(Q)\right|} \\ &\geq \frac{1}{|\mathcal{S}|} \sum_{Q \in \mathcal{S}} \sum_{j \in Q} \sum_{i \,:\, j \in \mathcal{O}_i(P^*)} \frac{w_i \, a_{ij}}{\left|\mathcal{O}_i(Q)\right|} \,. \end{aligned} \tag{5.19}$$

Now define, for each benchmark i, a parameter choice $J(i)$ from P^* of highest estimator value,

$$J(i) \;\in\; \underset{j \in P^*}{\operatorname{argmax}} \Big\{ E(A_{ij}) \Big\} \,,$$

where ties in the maximum estimator value are broken arbitrarily. Observe that when $J(i) \in Q$, the relationship $\mathcal{O}_i(Q) \subseteq \mathcal{O}_i(P^*)$ holds, since then both Q and P^* have the same highest estimator value (and $Q \subseteq P^*$). Thus when $J(i) \in Q$,

$$\left|\mathcal{O}_i(Q)\right| \;\leq\; \left|\mathcal{O}_i(P^*)\right| . \tag{5.20}$$

Returning to inequality (5.19), and applying relationship (5.20) in inequality (5.21) below,

$$
\begin{aligned}
f(P) \;&\geq\; \frac{1}{|\mathcal{S}|} \sum_{Q\,\in\,\mathcal{S}} \; \sum_{j\,\in\,Q} \; \sum_{i\,:\,j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(Q)\bigr|} \\
&=\; \frac{1}{|\mathcal{S}|} \sum_{i} \; \sum_{Q\,\in\,\mathcal{S}} \; \sum_{j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(Q)\bigr|} \\
&\geq\; \frac{1}{|\mathcal{S}|} \sum_{i} \; \sum_{Q\,\in\,\mathcal{S}\,:\,J(i)\,\in\,Q} \; \sum_{j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(Q)\bigr|} \\
&\geq\; \frac{1}{|\mathcal{S}|} \sum_{i} \; \sum_{Q\,\in\,\mathcal{S}\,:\,J(i)\,\in\,Q} \; \sum_{j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(P^*)\bigr|} \qquad (5.21) \\
&=\; \frac{1}{|\mathcal{S}|} \sum_{i} \Bigl| \bigl\{ Q\!\in\!\mathcal{S} \;:\; J(i)\!\in\!Q \bigr\} \Bigr| \sum_{j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(P^*)\bigr|} \\
&=\; \frac{\binom{\widetilde{k}-1}{\ell-1}}{\binom{\widetilde{k}}{\ell}} \; \sum_{i} \; \sum_{j\,\in\,\mathcal{O}_i(P^*)} \frac{w_i\, a_{ij}}{\bigl|\mathcal{O}_i(P^*)\bigr|} \\
&=\; \Bigl(\ell \big/ \widetilde{k}\Bigr)\; f(P^*) \\
&\geq\; \bigl(\ell/k\bigr)\; f(P^*)\,.
\end{aligned}
$$

Thus `Greedy` achieves approximation ratio at least ℓ/k.

Finally, to bound the running time of `Greedy`, consider an input instance with d features, n benchmarks, and m parameter choices in universe U. There are at most m^ℓ subsets of U of size at most ℓ, and evaluating objective function f on such a subset takes $O(d\ell n)$ time, so finding the optimal subset of size at most ℓ in the first step of `Greedy` takes $O(d\ell n m^\ell)$ time. The remaining for-loop considers at most k cardinalities, at most m parameter choices for each cardinality, and evaluates the objective function for each parameter choice on a subset of size at most k, which takes $O(dk^2mn)$ time. Thus the total time for `Greedy` is $O(d\ell n m^\ell + dk^2mn)$. For constant ℓ, this is polynomial time. □

In practice, we can compute optimal advisor sets of size up to $\ell = 5$ by exhaustive enumeration, as shown in Section 6.5.1. Finding an optimal advisor set of size $k = 10$, however, is currently far out of reach. Nevertheless, Theorem 6 shows we can still find reasonable approximations even for such large advisor sets, since for $\ell = 5$ and $k = 10$, `Greedy` is a (1/2)-approximation algorithm.

We next show it is not possible to prove a greater approximation ratio than in Theorem 6, as that ratio is in fact tight.

Theorem 7 (Tightness of Approximation Ratio) *The approximation ratio ℓ/k for algorithm* ***`Greedy`*** *is tight.*

Proof Since the ratio is obviously tight for $\ell = k$, assume $\ell < k$. For any arbitrary constant $0 < \delta < 1-(\ell/k)$, and for any error tolerance $0 \le \epsilon < 1$, consider the following infinite class of instances of Advisor Set with:

- benchmarks $1, 2, \ldots, n$,
- benchmark weights $w_i = 1/n$,
- cardinality bound $k = n$, and
- universe $U = \{0, 1, \ldots, n\}$ of $n+1$ parameter choices.

The estimator values for all benchmarks i are,

$$E(A_{ij}) \;=\; \begin{cases} 1, & j = 0; \\ (1-\epsilon)/2, & i = j > 0; \\ 0, & \text{otherwise}; \end{cases}$$

which can be achieved by appropriate feature vectors F_{ij}. The alternate alignment accuracies for all benchmarks i are,

$$a_{ij} \;=\; \begin{cases} (\ell/k) + \delta, & j = 0; \\ 1, & i = j > 0; \\ 0, & \text{otherwise}. \end{cases}$$

For such an instance of Advisor Set, an optimal set of size at most k is $P^* = \{1, \ldots, n\}$, which achieves $f(P^*) = 1$. Every optimal set $\widetilde{P}$ of size at most $\ell < k$ satisfies $\widetilde{P} \supseteq \{0\}$: it cannot include all of parameter choices $1, 2, \ldots, n$, so to avoid getting accuracy 0 on a benchmark it must contain parameter choice $j = 0$.

Moreover, every such set $\widetilde{P} \supseteq \{0\}$ has average accuracy $f\big(\widetilde{P}\big) = (\ell/k) + \delta$: parameter choice $j = 0$ has the maximum estimator value 1 on every benchmark, and no other parameter choice $j \neq 0$ has estimator value within ϵ of the maximum, so on every benchmark $\mathcal{A}_i\big(\widetilde{P}\big) = (\ell/k) + \delta$. Furthermore, every greedy augmentation $P \supseteq \widetilde{P}$ also has this same average accuracy $f(P) = f\big(\widetilde{P}\big)$. Thus on this instance the advisor set P output by `Greedy` has approximation ratio exactly

$$\frac{f(P)}{f(P^*)} \;=\; \frac{\ell}{k} \;+\; \delta\,.$$

Now suppose the approximation ratio from Theorem 6 is not tight, in other words, that an even better approximation ratio $\alpha > \ell/k$ holds. Then take $\delta = \big(\alpha - (\ell/k)\big)/2$, and run `Greedy` on the above input instance. On this instance, `Greedy` only achieves ratio

$$\frac{\ell}{k} \;+\; \delta \;=\; \frac{1}{2}\left(\frac{\ell}{k} \;+\; \alpha\right) \;<\; \alpha\,,$$

a contradiction. So the approximation ratio is tight. □

Summary

In this chapter, we have described an ILP for finding optimal advisor sets, and more general for finding an optimal advisor. As this ILP is not solvable in practice we further developed efficient approximation algorithm for finding estimator-aware advisor sets. In practice, we can find optimal oracle by solving a reduced ILP. Experiments with an implementation of the approximation algorithm on biological benchmarks, using various accuracy estimators from the literature, which are shown in Chapter 6, show it finds advisor sets that are surprisingly close to optimal. Furthermore, the resulting parameter advisors are significantly more accurate in practice than simply aligning with a single default parameter choice.

CHAPTER 6

Parameter Advising for `Opal`

Overview

In Chapters 1-5, we have described several approaches to constructing a parameter advisor. In this chapter, we demonstrate the performance of the trained advisor as learned on a set of benchmark alignments. We will also show the advisors performance compared to both the default parameter choice, as well as advisors learned on various accuracy estimators. We show `Facet` gives the best advising accuracy of any estimator currently available, and that by using estimator-aware advisor sets we can significantly increase the accuracy of the advisor over using oracle sets.

This chapter was adapted from portions of previous publications (DeBlasio et al., 2012b; Kececioglu and DeBlasio, 2013; DeBlasio and Kececioglu, 2014a, 2015).

6.1 Introduction

In characterizing six stages in constructing a multiple sequence alignment, Wheeler and Kececioglu (2007) gave as the first stage choosing the parameter values for the alignment scoring function. While many alignment tools allow the user to specify scoring function parameter values, such as affine gap penalties or substitution scoring matrices, typically only the default parameter values that the aligner provides are used. This default parameter choice is often tuned to optimize the average accuracy of the aligner over a collection of alignment benchmarks. While the default parameter values might be the single choice that works best on average on the benchmarks, for specific input sequences there may be a different choice on which the aligner outputs a much more accurate alignment.

This leads to the task of *parameter advising*: given particular sequences to align, and a set of possible parameter choices, recommend a parameter choice to the aligner that yields the most accurate alignment of those sequences. Parameter advising has three components: the set S of input sequences, the set P of parameter choices, and the aligner $\mathcal{A}$. (Here a *parameter choice* $p \in P$ is a vector $p = (p_1, \ldots, p_k)$ that specifies values for *all* free parameters in the alignment scoring function.) Given sequences S and parameter choice $p \in P$, we denote the alignment output by the aligner as $\mathcal{A}_p(S)$. Wheeler and Kececioglu (2007) call a procedure that takes the set of input sequences S and the set of parameter choices P, and outputs a parameter recommendation $p \in P$, an *advisor*. A perfect advisor, that always recommends the choice $p^* \in P$ that yields the highest accuracy alignment $\mathcal{A}_{p^*}(S)$, is called an *oracle*. In practice, constructing an oracle is impossible, since for any real set S of sequences that we want to align, a reference alignment for S is unknown (as otherwise we would not need to align them), so the true accuracy of any alignment of S cannot be determined. The concept of an oracle is useful, however, for measuring how well an actual advisor performs.

A natural approach for constructing a parameter advisor is to use an accuracy estimator E as a proxy for true accuracy, and recommend the parameter choice

$$\widetilde{p} \;:=\; \operatorname*{argmax}_{p \in P} \, E\big(\mathcal{A}_p(S)\big).$$

In its simplest realization, such an advisor will run the aligner $\mathcal{A}$ repeatedly on input S, once for each possible parameter choice $p \in P$, to select the output that has best estimated accuracy. Of course, to yield a quality advisor, this requires two ingredients: a good estimator E, and a good set P of parameter choices.

In Chapters 2 and 3 we presented our framework for accuracy estimation that lead to the new accuracy estimator `Facet` (short for "$\underline{\text{f}}$eature-based $\underline{\text{ac}}$curacy $\underline{\text{e}}$s$\underline{\text{t}}$imator"). Which is a linear combination of easy-to-compute feature functions of an alignment. We then went on in Chapter 5 to present a greedy approximation algorithm for finding advisor sets. Note that as discussed in Chapter 4, finding optimal advisor sets is NP-complete.

Given that we have the means to compute both accuracy estimators and advisor sets, we now apply all of this methodology to the task of parameter advising.

Plan of the chapter

In the next section, we describe the benchmarks that we use in all of our experiments. Recall that in order to learn both estimators and advisor sets, we must have examples for which we know the correct alignment and can calculate true accuracy. Section 6.3 shows examples of the estimator coefficients we learned, and compares our new `Facet` estimator to other estimators from the literature. Section 6.4 describes the differences between various methods for finding advisor sets. Section 6.5 assesses the increase in accuracy gained from parameter advising using `Facet` as well as other estimators. In addition, we show the increase in accuracy gained from using greedy advisor sets versus optimal oracle sets. Finally, the last section describes the software implementation of advising using `Facet` as a stand-alone tool, as an API, and within the `Opal` aligner.

6.2 Experimental methods

We evaluate our approach for deriving an accuracy estimator, and the quality of the resulting parameter advisor, through experiments on a collection of benchmark protein multiple sequence alignments. In these experiments, we compare parameter advisors that use our estimator and five other estimators from the literature: `COFFEE` (Notredame et al., 1998), `NorMD` (Thompson et al., 2001), `MOS` (Lassmann and Sonnhammer, 2005b), `HoT` (Landan and Graur, 2007), and `PredSP` (Ahola et al., 2008). (In terms of our earlier categorization of estimators, `COFFEE`, `NorMD` and `PredSP` are scoring-function-based, while `MOS` and `HoT` are support-based.) Other estimators from the literature that are not in this comparison group are: `AL2CO` (Pei and Grishin, 2001), which is known to be dominated by `NorMD` (see Lassmann and Sonnhammer, 2005b) `GUIDANCE` (Penn et al., 2010), which requires at least four sequences, and hence is not applicable to a large portion of the most challenging benchmarks in our study, as many hardest-to-align instances involve three very distant sequences; and `PSAR` (Kim and Ma, 2011), which at present is only implemented for DNA sequence alignments.

We refer to our estimator in the figures that follow by the acronym `Facet`, which is short for "feature-based accuracy estimator."

In our experiments, for the collection of alignment benchmarks we used the `BENCH` suite of Edgar (2009), which consists of 759 benchmarks, supplemented by a selection of 102 benchmarks from the `PALI` suite of Balaji et al. (2001). (`BENCH` itself is a selection of 759 benchmarks from (Bahr et al., 2001), `OxBench` (Raghava et al., 2003), and `SABRE` (Van Walle et al., 2005).) Both `BENCH` and `PALI` consist of protein multiple sequence alignments mainly induced by structural alignment of the known three-dimensional structures of the proteins. The entire benchmark collection consists of 861 reference alignments.

For the experiments, we measure the *difficulty* of a benchmark S by the true accuracy of the alignment computed by the multiple alignment tool `Opal` (Wheeler and Kececioglu, 2007, 2012) on sequences S using its default parameter choice, where the computed alignment is compared to the benchmark's reference alignment on its core columns. Using this measure, we binned the 861 benchmarks by difficulty, where we divided up the full range $[0, 1]$ of accuracies into 10 bins with difficulties $[(i-1)/10,\ i/10]$ for $i = 1, \ldots, 10$. As is common in benchmark suites, easy benchmarks are highly over-represented compared to hard benchmarks. The number of benchmarks falling in bins $[0.0, 0.1]$ through $[0.9, 1.0]$ are listed below.

bin	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
benchmarks	12	12	20	34	26	50	61	74	137	434

To correct for this bias in oversampling of easy benchmarks, our approaches for learning an estimator nonuniformly weight the training examples, as described earlier.

Notice that with this uniform weighting of bins, the singleton advising set P containing *only* the optimal default parameter choice will tend to an average advising accuracy $f(P)$ of 50% (illustrated later in Figures 6.2 and 6.3). This establishes, as a point of reference, average accuracy 50% as the *baseline* against which to compare advising performance.

Note that if we instead measure advising accuracy by uniformly averaging over *benchmarks*, then the predominance of easy benchmarks (for which little improvement is possible over the default parameter choice) makes both good and bad advisors tend to an average accuracy of nearly 100%. By uniformly averaging over *bins*, we can discriminate among advisors, though the average advising accuracies we report are now pulled down from 100% toward 50%.

For each reference alignment in our benchmark collection, we generated alternate multiple alignments of the sequences in the reference using `Opal` with varying parameter choices. `Opal` constructs multiple sequence alignments using as a building block the exact algorithm of Kececioglu and Starrett (2004) for optimally aligning two multiple alignments under the sum-of-pairs scoring function (Carrillo and Lipman, 1988) with affine gap penalties (Gotoh, 1982). Since `Opal` computes subalignments that are optimal with respect to a well-defined scoring function, it is an ideal testbed for evaluating parameter choices, and in particular parameter advising. Each *parameter choice* for `Opal` is a five-tuple $(\sigma, \gamma_I, \gamma_E, \lambda_I, \lambda_E)$ of parameter values, where σ specifies the amino acid substitution scoring matrix, pair γ_E, λ_E specifies the gap-open and gap-extension penalties for *external* gaps in the alignment (also called terminal gaps), and γ_I, λ_I specifies the gap penalties for *internal* gaps (or non-terminal gaps).

The universe U of parameter choices we consider in our experiments consists of over 2,000 such tuples $(\sigma, \gamma_I, \gamma_E, \lambda_I, \lambda_E)$. Universe U was generated as follows. For the substitution matrix σ, we considered matrices from the `BLOSUM` (Henikoff and Henikoff, 1992) and `VTML` (Müller et al., 2002) families. To accommodate a range of protein sequence divergences, we considered the following matrices from these families: $\{\texttt{BLSM45}, \texttt{BLSM62}, \texttt{BLSM80}\}$ and $\{\texttt{VTML20}, \texttt{VTML40}, \texttt{VTML80}, \texttt{VTML120}, \texttt{VTML200}\}$. For each of these eight matrices, we took the real-valued version of the similarity matrix and transformed it into a substitution cost matrix for `Opal` by negating, shifting, and scaling it to the range $[0, 100]$, and then rounding its entries to the nearest integer. For the gap penalties, we started from the default parameter setting for `Opal` (see Wheeler and Kececioglu, 2007), which is an optimal choice of gap penalties for the `BLSM62` matrix found by inverse parametric alignment (See Kececioglu and Kim, 2006; Kim and Kececioglu, 2008.) Around these default values we enumerated a Cartesian product of integer choices in the neighborhood of this central choice, generating over 2,100 four-tuples of gap penalties. The resulting set of roughly 16,900 parameter choices (each substitution matrix combined with each gap penalty assignment) was then reduced by examining the benchmarks in our collection as follows. In each hardness bin of benchmarks, we: (1) ran `Opal` with all of these parameter choices on the benchmarks in the bin, (2) for a given parameter choice measured the average accuracy of the alignments computed by `Opal` using that parameter choice on the bin, (3) sorted the parameter choices for a bin by their average accuracy, and (4) in each bin kept the top 25 choices with highest average accuracy. Unioning these top choices from all 10 hardness bins, and removing duplicates, gave our final set U. This universe U has 243 parameter choices.

To generate training and testing sets for our experiments on learning advisor sets, we used 12-*fold cross validation*. For each hardness bin, we evenly and randomly partitioned the benchmarks in the bin into twelve groups; we then formed twelve splits of the entire collection of benchmarks into a training class and a testing class, where each split placed one group in a bin into the testing class and the other eleven groups in the bin into the training class; finally, for each split we generated a *training set* and a *testing set* of example alignments as follows: for each benchmark B in a training or testing class, we generated $|U|$ example alignments in the respective training or testing set by running `Opal` on B with each parameter choice from U. An estimator learned on the examples from a training set was evaluated on examples from the corresponding testing set. The results we report are averages over twelve folds, where each *fold* is one of these pairs of associated training and testing sets. (Note that across the twelve folds, every example is tested on exactly once.) Each fold contains over 190,000 training examples.

When evaluating the `GUIDANCE` estimator, we used 4-fold cross validation on the reduced benchmark collection described earlier, with folds generated by the above procedure. Each of these folds has over 109,000 training examples.

6.3 Comparison of advisor estimators

To learn an estimator using the methods described in chapterch:estimator we must be given a set of alternate alignments produced by an aligner and their associated accuracy values. We use a set of alignment benchmarks that is a combination of the `BENCH` benchmark suite of Edgar (2009) supplemented with a subset of the `PALI` benchmark suite (Balaji et al., 2001). In total the benchmark set consisted of 861 benchmark alignments, for which we knew the correct alignment. We then computed an alignment for each of them using the `Opal` aligner using each of 16,896 parameter settings.

6.3.1 Finding an estimator

We found coefficients for the estimator using the difference-fitting method described in Section 2.3.2. We used only threshold-difference pairs with $\epsilon = 5\%$, for all 16,896 realignments of each benchmark. Note that here we found an estimator that is learned for pairs from all 861 benchmarks. When we use an estimator for experiments involving parameter advising, we use cross-validation to train new estimator coefficients for each fold, so as to not test on benchmarks that were used for training the estimator or advisor sets .

Of the features listed in Section 3.2, not all are equally informative, and some can weaken an estimator. When coefficients are found by solving the linear programs described in Chapter 2 on a set of example alignments some of the coefficients of the estimator will be zero. The best overall feature set found by this process is a 6-feature subset consisting of the following feature functions:

- Secondary Structure Agreement, $f_{\mathtt{SA}}$,
- Secondary Structure Blockiness, $f_{\mathtt{BL}}$,
- Secondary Structure Identity, $f_{\mathtt{SI}}$,
- Gap Open Density, $f_{\mathtt{GO}}$,
- Amino Acid Identity, $f_{\mathtt{AI}}$, and
- Core Column Percentage, $f_{\mathtt{CC}}$.

The corresponding fitted estimator is

$$\begin{aligned} E(A) \;=\;& 0.239\, f_{\mathtt{SA}}(A) \;+\; 0.141\, f_{\mathtt{BL}}(A) \;+\; 0.040\, f_{\mathtt{SI}}(A) \;+ \\ & 0.465\, f_{\mathtt{GO}}(A) \;+\; 0.204\, f_{\mathtt{AI}}(A) \;+\; 0.003\, f_{\mathtt{CC}}(A), \end{aligned}$$

Figure 3.1 shows a scatter plot of the five strongest features from the estimator. Notice that the feature with the highest coefficient value also has the smallest *range*.

6.3.2 Comparing estimators to true accuracy

To examine the fit of an estimator to true accuracy, the scatter plots in Figure 6.1 show the value of an estimator versus true accuracy on all example alignments in the 15-parameter test set. (This set has over 12,900 test examples. Note that these test examples are disjoint from the training examples used to fit our estimator.) The scatter plots show our `Facet` estimator as well as the `PredSP`, `MOS`, `COFFEE`, `HoT`, and `NorMD` estimators. We note that the `MOS` estimator, in distinction to the other estimators, receives as input *all* the alternate alignments of an example's sequences generated by the 15 parameter choices, which is much more information than is provided to the other estimators, which are only given the *one* example alignment.

An ideal estimator would be monotonic increasing in true accuracy. A real estimator approaches this ideal according to its *slope* and *spread*. To discriminate between low and high accuracy alignments for parameter advising, an estimator needs large slope with small spread. Comparing the scatter plots by spread, `Facet` and `PredSP` have the smallest spread; `MOS` and `COFFEE` have intermediate spread; and `HoT` and `NorMD` have the largest spread. Comparing by slope, `PredSP` and `NorMD` have the smallest slope; `Facet` and `HoT` have intermediate slope; and `MOS` and `COFFEE` have the largest slope. While `PredSP` has small spread, it also has small slope, which weakens its discriminative power. While `MOS` and `COFFEE` have large slope, they also have significant spread, weakening their discrimination. Finally `HoT` and `NorMD` have too large a spread to discriminate. Of all these estimators, `Facet` seems to achieve the best compromise of slope and spread, for a tighter monotonic trend across all accuracies. This better compromise between slope and spread may be what leads to improved performance for `Facet` on parameter advising, as demonstrated later in this section.

Our estimator combines six features to obtain its estimate. To give a sense of how these features behave, Figure 3.1 shows scatter plots of all of the feature functions' correlation with true accuracy (many which all use secondary structure). As noted in Section 6.3.1 the feature functions that we use for the `Facet` estimator are: Secondary Structure Agreement, Amino Acid Identity, Secondary Structure Blockiness, Secondary Structure Identity, and Core Column Percentage. Notice that the combined six-feature `Facet` estimator, shown in Figure 6.1, has smaller spread than any one of its individual features.

6.4 Comparison of advisor sets

Table 6.1 lists the parameter choices in the advisor sets found by the *greedy* approximation algorithm (augmenting from the optimal set of cardinality $\ell = 1$) for the `Opal` aligner with the `Facet` estimator for cardinalities $k \leq 20$, on one fold of training data. (The greedy sets vary slightly across folds.) In the table, the greedy set of cardinality k contains the parameter choices at rows 1 through k. (The entry at row 1 is the optimal *default parameter choice.*) Again a parameter choice is five-tuple $(\sigma, \gamma_I, \gamma_E, \lambda_I, \lambda_E)$, where γ_I and γ_E are gap-open penalties for non-terminal and terminal gaps respectively, and λ_I and λ_E are corresponding gap-extension penalties. The scores in the substitution matrix σ are dissimilarity values scaled to integers in the range $[0, 100]$. (The associated gap penalty values in a parameter choice relate to this range.) The accuracy column gives the average advising accuracy (in `Opal` using `Facet`) of the greedy set of cardinality k on *training* data, uniformly averaged over benchmark bins. Recall this averaging will tend to yield accuracies close to 50%.

Interestingly, while `BLOSUM62` Henikoff and Henikoff (1992) is the substitution

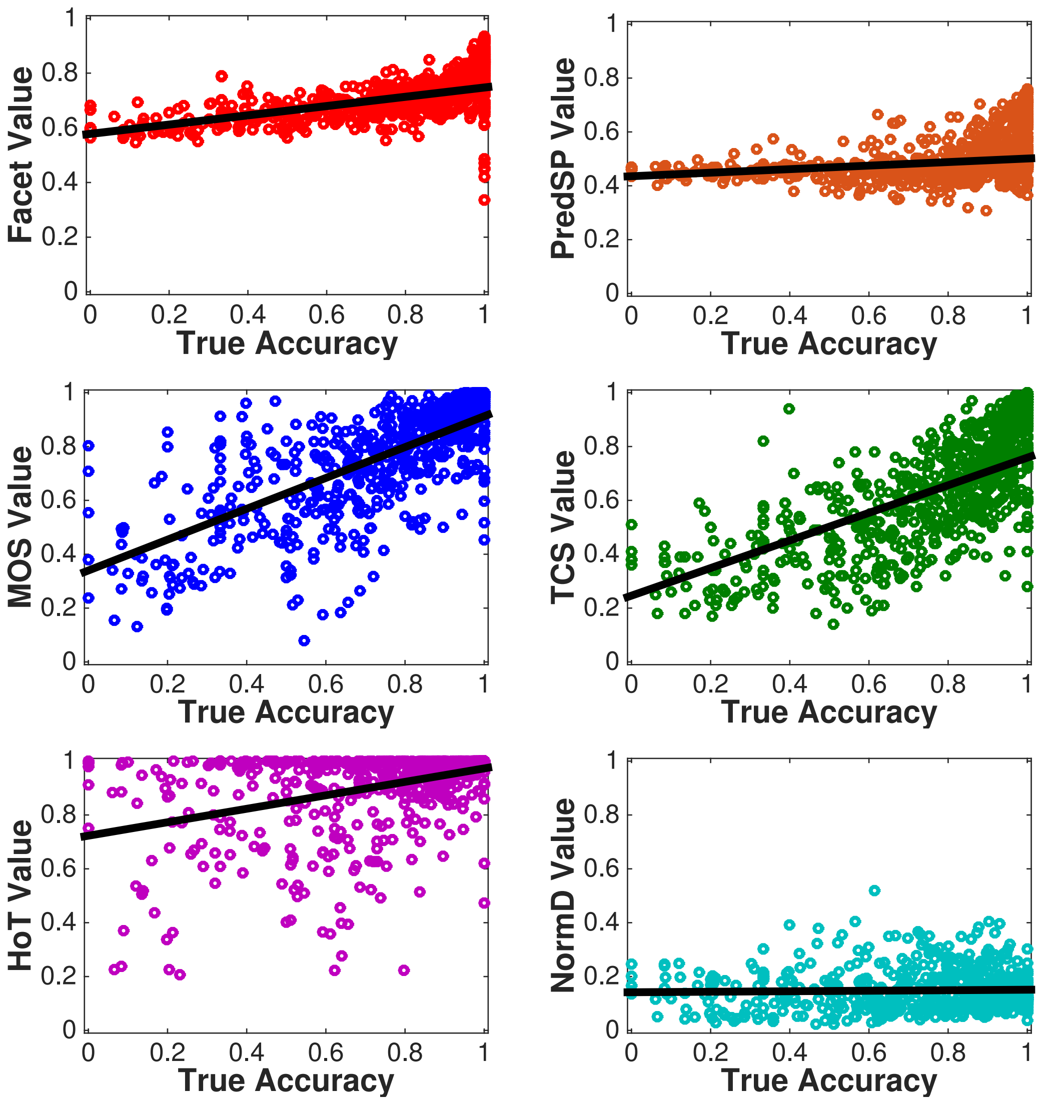


Figure 6.1: ***Correlation of estimators with accuracy.*** Each scatter plot shows the value of an estimator versus true accuracy for alignments of the 861 benchmarks used for testing aligned with the default parameter settings for the `Opal` aligner.

Table 6.1: ***Greedy Advisor Sets for*** `Opal` ***Using*** `Facet`

Cardinality k	Parameter choice $(\sigma,\ \gamma_I,\ \gamma_E,\ \lambda_I,\ \lambda_E)$	Average advising accuracy
1	$(\texttt{VTML200}, 50, 17, 41, 40)$	51.2%
2	$(\texttt{VTML200}, 55, 30, 45, 42)$	53.4%
3	$(\texttt{BLSUM80}, 60, 26, 43, 43)$	54.5%
4	$(\texttt{VTML200}, 60, 15, 41, 40)$	55.2%
5	$(\texttt{VTML200}, 55, 30, 41, 40)$	55.6%
6	$(\texttt{BLSUM45}, 65, 3, 44, 43)$	56.1%
7	$(\texttt{VTML120}, 50, 12, 42, 39)$	56.3%
8	$(\texttt{BLSUM45}, 65, 35, 44, 44)$	56.5%
9	$(\texttt{VTML200}, 45, 6, 41, 40)$	56.6%
10	$(\texttt{VTML120}, 55, 8, 40, 37)$	56.7%
11	$(\texttt{BLSUM62}, 80, 51, 43, 43)$	56.8%
12	$(\texttt{VTML120}, 50, 2, 45, 44)$	56.9%
13	$(\texttt{VTML200}, 45, 6, 40, 40)$	57.0%
14	$(\texttt{VTML40}, 50, 2, 40, 40)$	57.1%
15	$(\texttt{VTML200}, 50, 12, 43, 40)$	57.2%
16	$(\texttt{VTML200}, 45, 11, 42, 40)$	57.3%
17	$(\texttt{VTML120}, 60, 9, 40, 39)$	57.3%
18	$(\texttt{VTML40}, 50, 17, 40, 38)$	57.4%
19	$(\texttt{BLSUM80}, 70, 17, 42, 41)$	57.4%
20	$(\texttt{BLSUM80}, 60, 3, 42, 42)$	57.6%

scoring matrix most commonly used by standard aligners, it does not appear in a greedy set until cardinality $k = 11$. The `VTML` family Müller et al. (2002) appears more often than `BLOSUM`. The plateau in advising accuracy seen in earlier plots is also indicated in this training instance, though ever more gradual improvement remains as cardinality k increases.

6.4.1 Shared structure across advisor sets

To assess the similarity of advisor sets found by the three approaches considered in our experiments — *greedy sets* via the approximation algorithm, *exact sets* via exhaustive search, and *oracle sets* via integer linear programming — we examine their overlap both within and between folds.

Table 6.2 shows the composition of the greedy, exact, and oracle sets for the training instance in one fold, at cardinality $k = 2, 3, 4$ and tolerance $\epsilon = 0$. A non-

Table 6.2: ***Composition of Advisor Sets at Different Cardinalities*** k

Parameter choice $(\sigma,\ \gamma_I,\ \gamma_E,\ \lambda_I,\ \lambda_E)$	Advisor set Default	Greedy	Exact	Oracle
$k = 2$				
$(\texttt{VTML200}, 50, 17, 41, 40)$	(2)	(2)		
$(\texttt{VTML200}, 55, 30, 45, 42)$		(2)	(3)	(1)
$(\texttt{BLSUM80}, 60, 9, 43, 42)$			(2)	
$(\texttt{BLSUM45}, 65, 35, 44, 44)$				(3)
$k = 3$				
$(\texttt{VTML200}, 50, 17, 41, 40)$	(2)	(2)		
$(\texttt{VTML200}, 55, 30, 45, 42)$		(3)	(5)	(1)
$(\texttt{BLSUM80}, 60, 26, 43, 43)$		(2)	(2)	
$(\texttt{VTML200}, 55, 30, 41, 40)$			(6)	
$(\texttt{VTML40}, 45, 29, 40, 39)$				(7)
$(\texttt{BLSUM62}, 65, 16, 44, 42)$				(8)
$k = 4$				
$(\texttt{VTML200}, 50, 17, 41, 40)$	(2)	(2)		
$(\texttt{VTML200}, 55, 30, 45, 42)$		(3)	(9)	(6)
$(\texttt{BLSUM80}, 60, 26, 43, 43)$		(2)		
$(\texttt{VTML200}, 60, 15, 41, 40)$		(1)		
$(\texttt{VTML200}, 45, 6, 40, 40)$			(8)	(1)
$(\texttt{VTML200}, 55, 30, 41, 40)$			(8)	
$(\texttt{BLSUM80}, 55, 19, 43, 42)$			(1)	
$(\texttt{VTML40}, 45, 29, 40, 39)$				(4)
$(\texttt{BLSUM62}, 65, 35, 44, 42)$				(3)

Table 6.3: ***Number of Folds Where Greedy and Exact Sets Share Parameters***

Intersection cardinality	Advisor set cardinality $k = 2$	$k = 3$	$k = 4$	$k = 5$
0	9	4	3	2
1	3	5	6	5
2	0	3	3	4
3		0	0	1
4			0	0
5				0

blank entry in the table indicates that the parameter choice at its row is contained in the advisor set at its column. (The column labeled "default" indicates the optimal *default parameter choice* for the fold, or equivalently, the exact set of cardinality $k = 1$.) The value in parentheses at an entry is the number of folds (for twelve-fold cross-validation) where that parameter choice appears in that advisor set. (For example, at cardinality $k = 4$, the second parameter choice ($\texttt{VTML200}, 55, 30, 45, 42$) is in the greedy, exact, and oracle sets for this particular fold, and overall is in exact sets for 9 of 12 folds, including this fold.) Surprisingly, the default parameter choice (the best single choice) never appears in the exact or oracle sets for this fold at any of the cardinalities beyond $k = 1$, and also is reused as the default in only one other fold. In general there is relatively little overlap between these advisor sets: often just one and at most two parameter choices are shared.

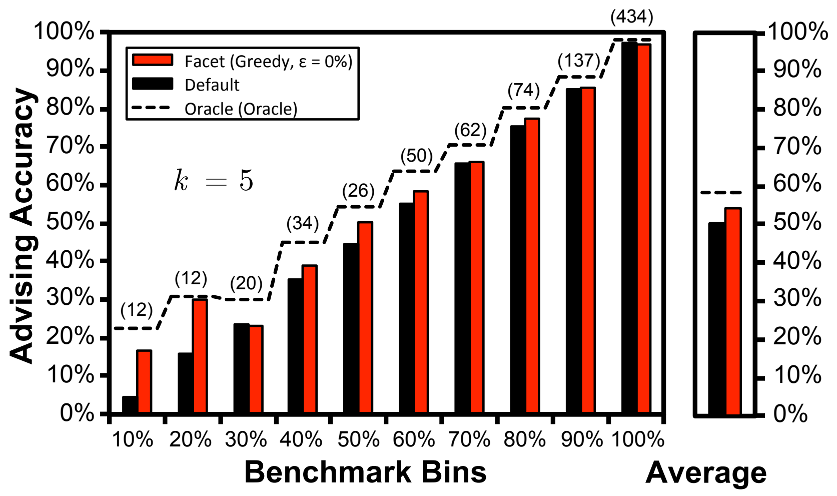
Table 6.3 examines whether this trend continues at other folds, by counting how many training instances (out of the twelve folds) share a specified number of parameter choices between their *greedy* and *exact* sets, for a given advisor set cardinality k. (For example, at cardinality $k = 4$, six training instances share exactly one parameter choice between their greedy and exact sets; in fact, the fold shown in Table 6.2 is one such instance.) On the whole, the two "estimator-aware" advisor sets — the greedy and exact sets — are relatively dissimilar, and never share more than $\lceil k/2 \rceil$ parameter choices.

6.5 Application to parameter advising

Given the accuracy estimator learned using difference fitting that we have described in earlier sections, and the advisor sets described in the previous section, we now evaluate the advising accuracy of our new parameter advisor.

6.5.1 Learning advisor sets by different approaches

We first study the advising accuracy of parameter sets learned for the `Facet` estimator by different approaches. Our protocol began by constructing an optimal *oracle*



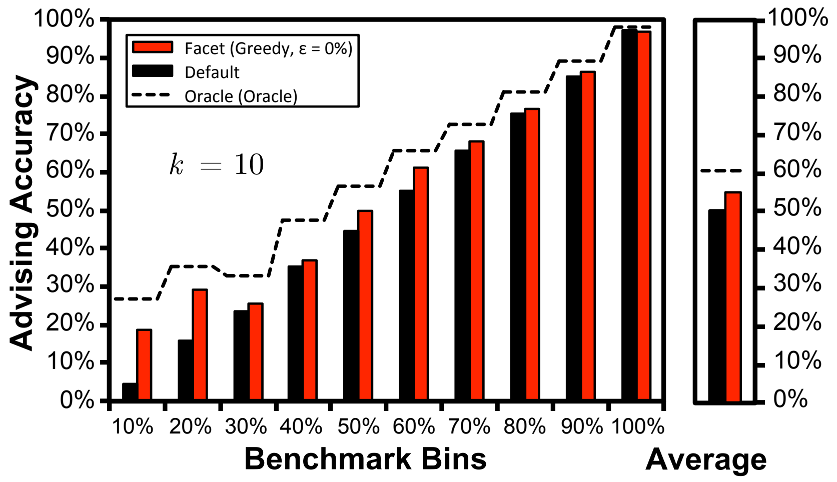


Figure 6.2: ***Advising accuracy of*** `Facet` ***within benchmark bins.*** These bar charts show the advising accuracy of various approaches to finding advisor sets, for cardinality $k = 5, 10$. For each cardinality, the horizontal axis of the chart on the left corresponds to benchmark bins, and the vertical bars show advising accuracy averaged over the benchmarks in each bin. Black bars give the accuracy of the optimal *default* parameter choice, and red bars give the accuracy of advising with `Facet` using the *greedy* set. The dashed line shows the limiting performance of a perfect advisor: an oracle with true accuracy as its estimator using an optimal *oracle* set. In the top chart, the numbers in parentheses above the bars are the number of benchmarks in each bin. The narrow bar charts on the right show advising accuracy uniformly averaged over the bins.

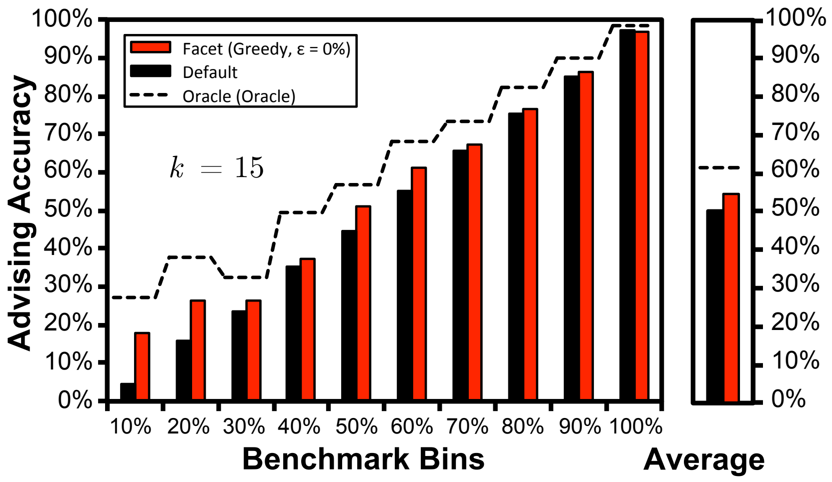


Figure 6.3: ***Advising accuracy of*** `Facet` ***within benchmark bins.*** These bar charts show the advising accuracy of various approaches to finding advisor sets, for cardinality $k = 15$. For each cardinality, the horizontal axis of the chart on the left corresponds to benchmark bins, and the vertical bars show advising accuracy averaged over the benchmarks in each bin. Black bars give the accuracy of the optimal *default* parameter choice, and red bars give the accuracy of advising with `Facet` using the *greedy* set. The dashed line shows the limiting performance of a perfect advisor: an oracle with true accuracy as its estimator using an optimal *oracle* set. In the top chart, the numbers in parentheses above the bars are the number of benchmarks in each bin. The narrow bar charts on the right show advising accuracy uniformly averaged over the bins.

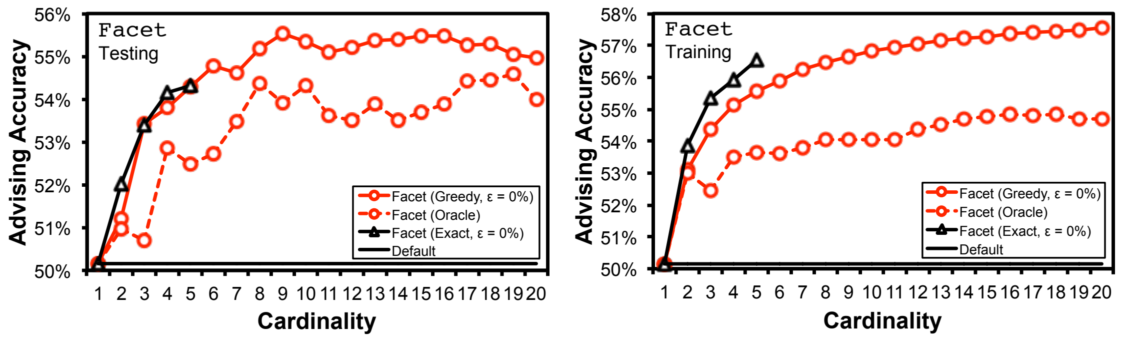


Figure 6.4: ***Advising using exact, greedy, and oracle sets with*** `Facet`**.** The plots show advising accuracy using the `Facet` estimator with parameter sets learned by the optimal *exact* algorithm and the *greedy* approximation algorithm for Advisor Set, and with *oracle* sets. The horizontal axis is the cardinality of the advisor set, while the vertical axis is the advising accuracy averaged over the benchmarks. Exact sets are known only for cardinalities $k \leq 5$; greedy sets are augmented from the exact set of cardinality $\ell = 1$. The left and right plots show accuracy on the testing and training data, respectively, where accuracies are averaged over all testing or training folds.

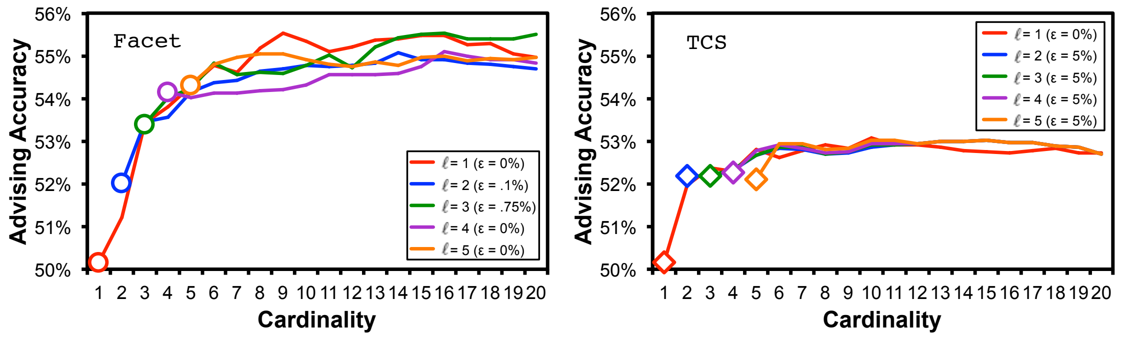


Figure 6.5: ***Greedily augmenting exact advisor sets.*** The left and right plots show advising accuracy using the `Facet` and `TCS` estimators respectively, with advisor sets learned by procedure `Greedy`, which augments an exact set of cardinality ℓ to form a larger set of cardinality $k > \ell$. Each curve is greedily augmenting from a different exact cardinality ℓ. The horizontal axis is the cardinality k of the augmented set; the vertical axis is advising accuracy on testing data, averaged over all benchmarks and all folds.

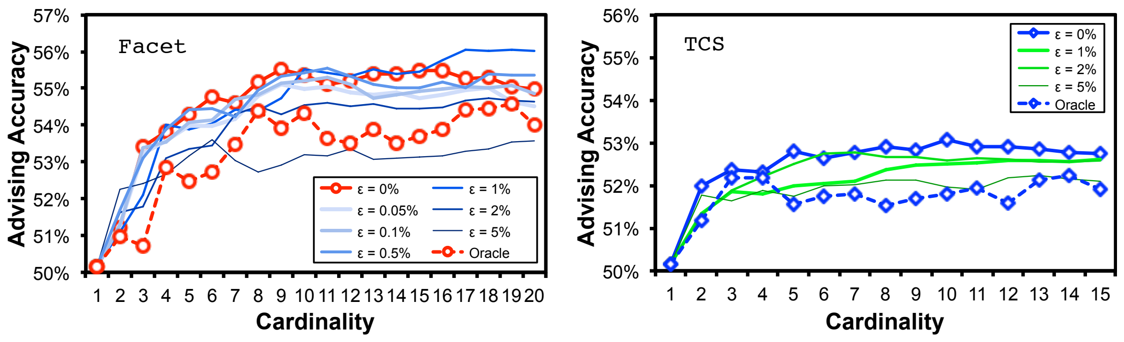


Figure 6.6: ***Effect of error tolerance on advising accuracy using greedy sets.*** The plots show advising accuracy on testing data using greedy sets learned for the two best estimators, `Facet` and `TCS`, at various error tolerances $\epsilon \geq 0$. The plots on the left and right are for `Facet` and `TCS`, respectively. For comparison, both plots also include a curve showing performance using the estimator on oracle sets, drawn with a dashed line. The solid curves with circles and diamonds highlight the best overall error tolerance of $\epsilon = 0$.

set for cardinalities $1 \leq k \leq 20$ for each training instance. A coefficient vector for the advisor's estimator was then found for each of these oracle sets by the difference-fitting method described in Kececioglu and DeBlasio (2013). Using this estimator learned for the training data, exhaustive search was done to find optimal *exact* advisor sets for cardinalities $k \leq 5$. The optimal exact set of size $\ell = 1$ (the best default parameter choice) was then used as the starting point to find near-optimal *greedy* advisor sets by our approximation algorithm for $k \leq 20$. Each of these advisors (an advising set combined with the estimator) was then used for parameter advising in `Opal`, returning the computed alignment with highest estimator value. These set-finding approaches are compared based on the accuracy of the alignment chosen by the advisor, averaged across bins.

Figure 6.4 shows the performance of these advisor sets under twelve-fold cross validation. The left plot shows advising accuracy on the testing data averaged over the folds, while the right plot shows this on the training data.

Notice that while there is a drop in accuracy when an advising set learned using the greedy and exact methods is applied to the testing data, the drop in accuracy is greatest for the exact sets. The value of ϵ shown in the plot maximizes the accuracy of the resulting advisor on the testing data. Notice also that for cardinality $k \leq 5$ (for which exact sets could be computed), on the testing data the greedy sets are often performing as well as the optimal exact sets.

Figures 6.2 and 6.3 shows the performance within each benchmark bin when advising with `Facet` using greedy sets of cardinality $k = 5, 10, 15$ ($k = 5$ and 10 in Figure 6.2 top and bottom respectively, $k = 15$ in Figure 6.3) Notice that for many bins, the performance is close to the best-possible accuracy attainable by any advisor, shown by the dashed line for a perfect oracle advisor. The greatest boost over the default parameter choice is achieved on the bottom bins that contain the hardest benchmarks.

6.5.2 Varying the exact set for the greedy algorithm

To find the appropriate cardinality ℓ of the initial exact solution that is augmented within approximation algorithm `Greedy`, we examined the advising accuracy of the greedy sets learned when using cardinalities $1 \leq \ell \leq 5$. Figure 6.5 shows the accuracy of the resulting advisor using greedy sets of cardinality $1 \leq k \leq 20$, augmented from exact sets of cardinality $1 \leq \ell \leq 5$, using for the estimator both `Facet` and `TCS`. (These are the two best estimators, as discussed in Section 6.5.4 below). The points plotted with circles show the accuracy of the optimal exact set that is used within procedure `Greedy` for augmentation.

Notice that the initial exact set size ℓ has relatively little effect on the accuracy of the resulting advisor; at most cardinalities, starting from the single best parameter choice ($\ell = 1$) has highest advising accuracy. This is likely due to the behavior observed earlier in Figure 6.4, namely that exact sets do not generalize as well as greedy sets.

6.5.3 Varying the error tolerance for the greedy algorithm

When showing experimental results, an error tolerance ϵ has always been used that yields the most accurate advisor on the testing data. Prior to conducting these

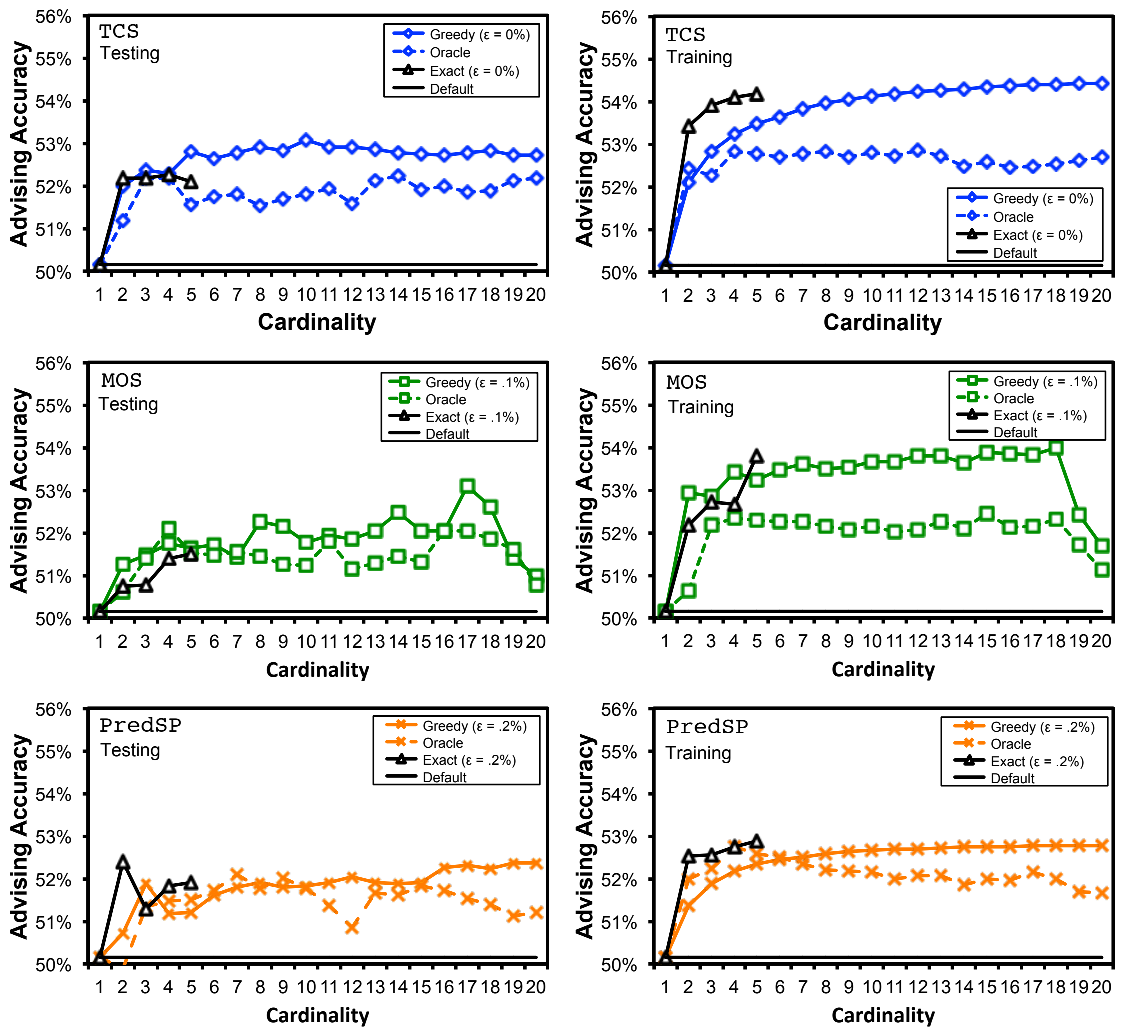


Figure 6.7: ***Comparing testing and training accuracies of various estimators.*** The plots show the advising accuracies on testing and training data using `TCS`, `MOS`, and `PredSP` with parameter sets learned for these estimators by the *exact* and *greedy* algorithms for Advisor Set, and with *oracle* sets. From top to bottom, the estimators used are `TCS`, `MOS`, and `PredSP`, with *testing* data plotted on the left, and *training* data on the right.

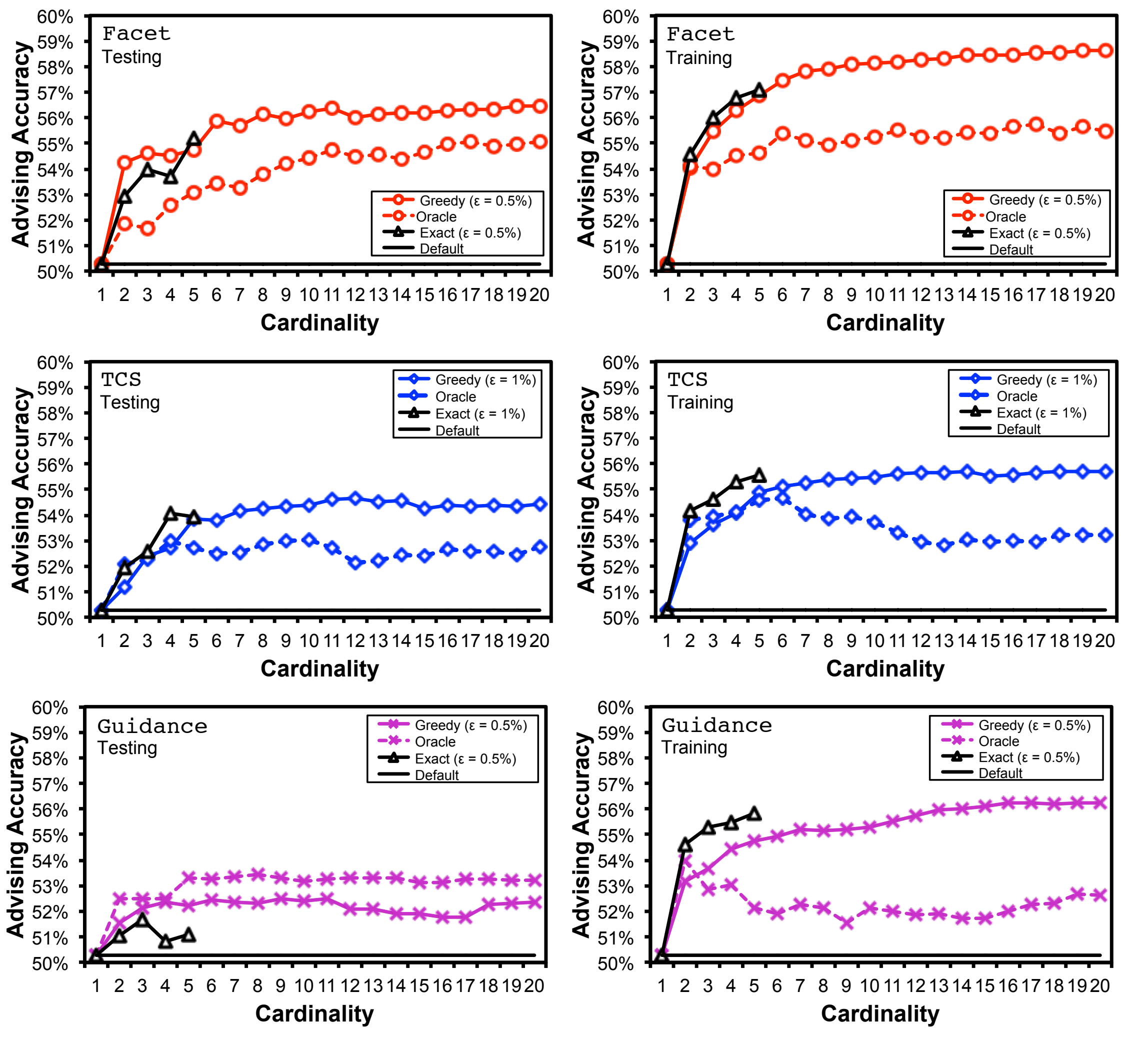


Figure 6.8: ***Comparing testing and training accuracies of estimators on benchmarks with at least four sequences.*** The plots show advising accuracies for *testing* and *training* data on benchmarks with at least four sequences, using `Facet`, `TCS`, and `GUIDANCE` with *exact*, *greedy*, and *oracle* sets.

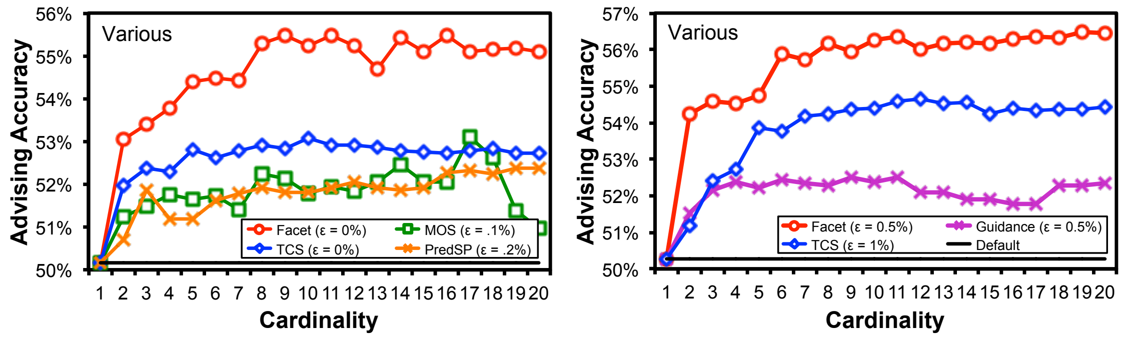


Figure 6.9: ***Comparing all estimators on greedy advisor sets.*** The plots show advising accuracy on *greedy* sets learned for the estimators `Facet`, `TCS`, `MOS`, `PredSP`, and `GUIDANCE`. The vertical axis is advising accuracy on *testing* data, averaged over all benchmarks and all folds. The horizontal axis is the cardinality k of the greedy advisor set. Greedy sets are augmented from the exact set of cardinality $\ell = 1$. The plot on the left uses the *full* benchmark suite; the plot on the right, which includes `GUIDANCE`, uses a *reduced* suite of all benchmarks with at least four sequences.

experiments, our expectation was that a nonzero error tolerance $\epsilon > 0$ would boost the generalization of advisor sets. Figure 6.6 shows the effect of different values of ϵ on the testing accuracy of an advisor using greedy sets learned for the `Facet` and `TCS` estimators. (While the same values of ϵ were tried for both estimators, raw `TCS` scores are integers in the range $[0, 100]$ which were scaled to real values in the range $[0, 1]$, so for `TCS` any $\epsilon < 0.1$ is equivalent to $\epsilon = 0$.) No clear relationship between testing accuracy and error tolerance is evident, though for `Facet` and `TCS` alike, setting $\epsilon = 0$ generally gives the best overall advising accuracy.

6.5.4 Learning advisor sets for different estimators

In addition to learning advisor sets for `Facet` (Kececioglu and DeBlasio, 2013), we also learned sets for the best accuracy estimators from the literature: namely, `TCS` (Chang et al., 2014), `MOS` (Lassmann and Sonnhammer, 2005b), `PredSP` (Ahola et al., 2008), and `GUIDANCE` (Penn et al., 2010). The scoring-function-based accuracy estimators `TCS`, `PredSP`, and `GUIDANCE` do not have any dependence on the advisor set cardinality or the training benchmarks used. The support-based estimator `MOS`, however, requires a set of alternate alignments in order to compute its estimator value on an alignment. In each experiment, an alignment's `MOS` value was computed using alternate alignments generated by aligning under the parameter choices in the oracle set; if the parameter choice being tested on was in the oracle set, it was removed from this collection of alternate alignments.

After computing the values of these estimators, exhaustive search was used to find optimal exact sets of cardinality $\ell \leq 5$ for each estimator, as well as greedy sets of cardinality $k \leq 20$ (augmenting from the exact set for $\ell = 1$).

The tendency of exact advisor sets to not generalize well is even more pronounced when accuracy estimators other than `Facet` are used. Figure 6.7 shows the performance on testing and training data of greedy, exact, and oracle advisor sets learned for the best three other estimators: `TCS`, `MOS`, and `PredSP`. The results for greedy advisor sets for `TCS` at cardinalities larger than 5 have similar trend to those seen for `Facet` (with now a roughly 1% accuracy improvement over the oracle set), but surprisingly with `TCS` its exact set always has lower testing accuracy than its greedy set. Interestingly, for `MOS` its exact set rarely has better advising accuracy than the oracle set. For `PredSP`, at most cardinalities (with the exception of $k = 3$) the exact set has higher accuracy than the greedy set on testing data, though this is offset by the low accuracy of the estimator.

We also tested `GUIDANCE`, `Facet`, and `TCS` on the reduced suite of all benchmarks with at least four sequences (as required by `GUIDANCE`). Figure 6.8 shows the advising accuracy of set-finding methods using these estimators on these benchmarks. Notice that on this reduced suite the results generally stay the same, though for `Facet` there is more of a drop in performance of the exact set from training to testing, and the set found by `Greedy` generally has greater accuracy on the reduced suite than the full suite.

Finally, a complete comparison of the advising performance of *all estimators* using greedy sets is shown in Figure 6.9. (The plot on the right shows advising accuracy on testing data for `GUIDANCE`, `Facet`, and `TCS` on the reduced suite of benchmarks with at least four sequences.) Advising with each of these estimators tends to eventually reach an accuracy plateau, though their performance is always boosted by using advisor sets larger than a singleton default choice. The plateau for `Facet` (the top curve in the plots) generally occurs at the greatest cardinality and accuracy.

6.6 Software

Parameter advising in our software implementation can be performed in one of two ways:

(1) ***`Facet` aligner wrapper*** – Similar to using `Facet` on the command line, you can use a set of provided `Perl` scripts that runs `PSIPRED` to predict the protein secondary structure, uses a provided set of `Opal` parameter settings, computes alignments for each of these settings, computes the `Facet` score, and identifies the highest accuracy alignment. The script must be configured for each user's installation location of `Opal` and `PSIPRED`.

(2) ***Within `Opal`*** – The newest version of the `Opal` aligner can perform parameter advising internally. The advising set is given to `Opal` using the `--advising_configuration_file` command line argument. The most accurate alignment will then be output to the file identified by the `--out` argument. More details of the parameter advising modifications made to `Opal` are given in Section 6.6.1.

The advising wrapper as well as oracle and greedy sets, can be found on `http://facet.cs.arizona.edu`.

6.6.1 `Opal` version 3

We have updated the `Opal` aligner to include parameter advising inside the aligner. `Opal` can now construct alignments under various configuration settings *in parallel* to attempt to come close to producing a parameter-advised alignment in no more wall-time than aligning under a single default parameter.

Because both `Opal` and `Facet` are implemented in `Java`, they can be integrated easily. When an alignment is constructed, a `Facet` score is automatically generated if secondary structure labeling is given. The secondary structure can be generated using a wrapper for `PSIPRED`, which will run the secondary-structure prediction and then format the output so it is readable by `Opal`. For the structure to be able to be used for `Facet`, you must input this file using the `--facet_structure` command-line argument. This score is output to standard out. In addition, the score can be printed into the file name by adding the string `__FACETSCORE__` to the output file name argument, when the file is created this string is replaced with the computed `Facet` score.

The new version of `Opal` also includes the ability to use popular versions of the `PAM`, `BLOSUM` and `VTML` matrices. These can be specified via the `--cost` command line argument. If the specified cost name is not built in, you can specify a new matrix by giving the file name via the same command line argument. The matrix file should fallow the same formatting convention as `BLAST` matrices.

If an advisor set of parameter settings is specified using the `--advisor_configuration_file` command line argument then `Opal` will construct an alignment for each of the configurations in the file. If in addition a secondary structure prediction is specified, `Opal` will perform parameter advising. The input advisor set contains a list of parameter settings in 5-tuple format mentioned earlier ($\sigma.\gamma_I.\gamma_T.\lambda_I.\lambda_T$, where σ is the replacement matrix, γ_I and γ_T specify the internal and terminal gap extension penalties and λ_I and λ_T specify the gap open penalties). If advising is performed, the alignment with the highest estimated accuracy is output to the file specified in the `--out_best` command line argument. In addition, `Opal` can output the results for each of the configurations specified in the advisor set using `--out_config`; the filename in that case should contain the string `__CONFIG__`, which will then be replaced with the parameter setting.

While an alignment must be generated for each parameter setting in the advising set, the construction of these alignments is independent. Because of this we enabled

`Opal` to construct the alignments in the advising set in parallel. `Opal` will automatically detect how many processors are available and run that many threads to construct alignments, but this can be overridden by specifying a maximum number of threads using the `--max_threads` command line argument. By doing this, if the number of processors available is larger than the number of parameter choices in the advising set, then the total wall-clock time is close to the time it would take to run the multiple sequence alignment of the input using just a single default parameter choice.

Version 3.0 of the `Opal` aligner is available at `http://opal.cs.arizona.edu`, and the development version of `Opal` is available on `GiHub` at `http://git.io/Opal`.

Summary

In this chapter, we described our experimental methodology for testing the advising accuracy of the `Facet` estimator, as well as demonstrated the resulting increase in advising accuracy over using a single default parameter choice.

CHAPTER 7

Aligner Advising for Ensemble Alignment

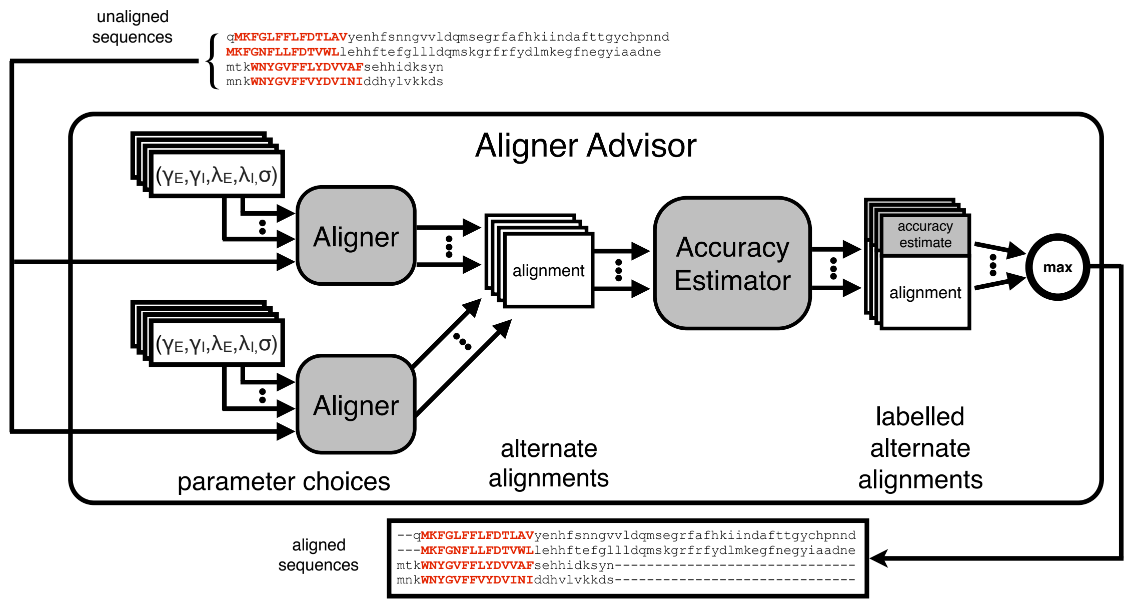
Overview

The multiple sequence alignments computed by an aligner for different *settings* of its parameters, as well as the alignments computed by different *aligners* using their default settings, can differ markedly in accuracy. *Parameter advising* is the task of choosing a parameter setting for an aligner to maximize the accuracy of the resulting alignment. We extend parameter advising to *aligner advising*, which in contrast chooses among a set of aligners to maximize accuracy. In the context of aligner advising, *default* advising selects from a set of aligners that are using their default settings, while *general* advising selects both the aligner and its parameter setting.

In this chapter, we apply aligner advising for the first time, to create a true *ensemble aligner*. Through cross-validation experiments on benchmark protein sequence alignments, we show that parameter advising boosts an aligner's accuracy beyond its default setting for virtually all of the standard aligners currently used in practice. Furthermore, aligner advising with a collection of aligners further improves upon parameter advising with any single aligner, though surprisingly the performance of default advising on testing data is actually superior to general advising due to less overfitting to training data.

The new ensemble aligner that results from aligner advising is significantly more accurate than the best single default aligner, especially on hard-to-align sequences. This successfully demonstrates how to construct out of a collection of individual aligners, a more accurate ensemble aligner.

This chapter was adapted from portions of a previous publication (DeBlasio and Kececioglu, 2015).



```
qMKFGLFFLFDTLAVyenhfsnngvvldqmsegrfafhkiindafttgychpnnd
MKFGNFLLFDTVWLlehhftefgllldqmskgrfrfydlmkegfnegyiaadne
mtkWNYGVFFLYDVVAFsehhidksyn
mnkWNYGVFFVYDVINIddhylvkkds
```

```
--qMKFGLFFLFDTLAVyenhfsnngvvldqmsegrfafhkiindafttgychpnnd
---MKFGNFLLFDTVWLlehhftefgllldqmskgrfrfydlmkegfnegyiaadne
mtkWNYGVFFLYDVVAFsehhidksyn------------------------------
mnkWNYGVFFVYDVINIddhvlvkkds------------------------------
```

Figure 7.1: ***Overview of the ensemble alignment process.*** An advisor set is a collection of aligners and associated parameter choices. Default aligner advising sets only contain aligners and their default parameter settings, while general aligner advising sets can include non-default parameter settings. An accuracy estimator labels each candidate alignment with an accuracy estimate. (Conceptually, an oracle gives the true accuracy of an alignment.) The alignment with the highest estimated accuracy is chosen by the advisor.

7.1 Introduction

While it has long been known that the multiple sequence alignment computed by an aligner strongly depends on the settings for its tunable parameters, and that different aligners using their default settings can output markedly different alignments of the same input sequences, there has been relatively little work on how to automatically choose the best parameter settings for an aligner, or the best aligner to invoke, to obtain the most accurate alignment of a given set of input sequences.

Automatically choosing the best parameter setting for an aligner on a given input was termed by Wheeler and Kececioglu (2007), *parameter advising*. In their framework, an advisor takes a *set* of parameter settings, together with an *estimator* that estimates the accuracy of a computed alignment, and invokes the aligner on each setting, evaluates the accuracy estimator on each resulting alignment, and chooses

```
d1flma    8    ... fevlknegvvAIATQgedgphlvntwnsylkv-ldgnrivvpvggmhkteanva-rde ...  63
d1ci0a    18   ... tkw-fn--------eakedpret-------------lpeaiTFSS-------Aelpsg ...  46
d1nrga    16   ... aaw-fe--------eavqcpdig-------------eanamCLAT-------Ct-rdg ...  43
d1ejea    22   ... hriltprptvMVTTVdeegninaapfsftmpvsidppvvafasapdhhtarnie-sth ...  78
d1i0ra    8    ... ykisyglyIVTSEsngrkcgqiant---vfqltskpvqiavclnkendthnavk-esg ...  61
```

(a) Lower-accuracy alignment computed by MUMMALS

```
d1flma    1    ... ------mlpgtffevlkne-----gvvAIATQg-edgph--lvntwnsylk---vldg ...  41
d1ci0a    12   ... d-dpidlftkwfneakedpretlpeaiTFSSAelpsgr----vssrillfk---eldh ...  59
d1nrga    9    ... sldpvkqfaawfeeavqcpdigeanamCLATCt-rdgk----psarmlllk---gfgk ...  56
d1ejea    11   ... s-mdfedfpvesahriltpr----ptvMVTTVd-eegn----inaapfsftmpvsidp ...  56
d1i0ra    1    ... --mdveafykisy------------glyIVTSE-sngrkcgqiantvfqlt---s-kp ...  39
```

(b) Higher-accuracy alignment computed by Opal

Figure 7.2: ***Aligner choice affects the accuracy of computed alignments.*** (a) Part of an alignment of benchmark sup_125 from the SABRE (Van Walle et al., 2005) suite computed by MUMMALS (Pei and Grishin, 2006) using its default parameter choice; this alignment has accuracy value 28.9%, and Facet estimator value 0.540. (b) Alignment of the same benchmark by Opal (Wheeler and Kececioglu, 2007) using its default parameter choice, which has 49.9% accuracy, and higher Facet value 0.578. In both alignments, the positions that correspond to *core blocks* of the reference alignment, which should be aligned in a correct alignment, are highlighted in bold.

the setting that gives the alignment of highest estimated accuracy. Analogously, we call automatically choosing the best aligner for a given input, *aligner advising.* Figure 7.1 shows an overview of aligner advising. Notice that compared to the similar Figure 1.3 which describes the parameter advising figure the aligner has been moved into the advisor set and the advisor may use more than one aligner to produce alternate alignments

To make this concrete, Figure 7.2 shows an example of advising on a benchmark set of protein sequences for which a correct reference alignment is known, and hence for which the true accuracy of a computed alignment can be determined. In this example, the Facet estimator is used to estimate the accuracy of two alignments computed by the Opal (Wheeler and Kececioglu, 2012) and MUMMALS (Pei and Grishin, 2006) aligners. For these two alignments, the one of higher Facet value has higher true accuracy as well, so an advisor armed with the Facet estimator would in fact output the more accurate alignment to a user.

For a collection of aligners, this kind of advising is akin to an *ensemble* approach to alignment, which selects a solution from those output by different methods to obtain in effect a new method that ideally is better than any individual method. Ensemble methods have been studied in machine learning (Zhihua, 2012), which combine the results of different classifiers to produce a single output classification. Typically such ensemble methods from machine learning select a result by *voting*. In contrast, an advisor combines the results of aligners by selecting one via an *estimator*.

In this chapter, we extend the framework of parameter advising to aligner advising, and obtain by this natural approach a true *ensemble aligner*. Moreover as our experimental results show, the resulting ensemble aligner is significantly more accurate than any individual aligner.

7.1.1 Related work

Wheeler and Kececioglu (2007) first introduced the notion of *parameter advisors*; Kececioglu and DeBlasio (2013) investigated the construction of alignment *accuracy estimators*, resulting in the `Facet` estimator (DeBlasio et al., 2012b; DeBlasio and Kececioglu, 2014b); DeBlasio and Kececioglu (2014a, 2016) investigated how to best form the set of parameter choices for an advisor, called an *advisor set*, developing an efficient approximation algorithm for finding a near-optimal advisor set for a given estimator. This prior work applied parameter advising to boosting the accuracy of the `Opal` aligner (Wheeler and Kececioglu, 2012). In contrast, this chapter applies *parameter advising* to all commonly-used aligners, and *aligner advising* to combine them into a new, more accurate, ensemble aligner.

To our knowledge, the only prior work on combining aligners is by Wallace et al. (2006) on `M-Coffee`, and by Muller et al. (2010) on `AQUA`. The `AQUA` tool chooses between an alignment computed by `Muscle` (Edgar, 2004b) or `MAFFT` (Katoh et al., 2005) based on their `NorMD` (Thompson et al., 2001) score; our results given in Chapter 6 show that for choosing the more accurate alignment, the `NorMD` score used by `AQUA` is much weaker than the `Facet` estimator used here for aligner advising. `M-Coffee` uses a standard progressive alignment heuristic to compute an alignment under position-dependent substitution scores whose values are determined by alignments from different aligners. As Section 7.3.3 later shows, when run on the same set of aligners, `M-Coffee` is strongly dominated by the ensemble approach of this chapter.

Contributions

Our prior work on parameter advising focused on boosting the accuracy of the `Opal` aligner (Wheeler and Kececioglu, 2007, 2012) through an input-dependent choice of parameter values. This chapter applies our advising technique for the first time to aligners *other than* `Opal`, both by advising parameter choices for them, and by advising how to combine them into an new ensemble aligner.

Plan of the chapter

An advisor selects aligners and parameter values from a small set of choices that is drawn from a larger universe of all possible choices. Section 7.2 describes how we construct this universe of aligners and their parameter choices for advisor learning. Section 7.3 then experimentally evaluates our approach to ensemble alignment on real biological benchmarks. Finally, Section 7.4 gives conclusions, and offers directions for further research.

7.2 Constructing the universe for aligner advising

We extend *parameter advising* with a single aligner to *aligner advising* with a collection of aligners, by having the choices in the advisor set now specify both a particular aligner and a parameter setting for that aligner. To specify the *universe* that such an advisor set is drawn from during learning, we must determine what aligners to consider, and what parameter settings to consider for those aligners.

7.2.1 Determining the universe of aligners

For *default aligner advising*, where the advisor set consists of distinct aligners, each using their default parameter setting, we learned advisor sets over a universe containing as many of the commonly-used aligners from the literature as possible. Specifically, the universe for default advising consisted of the following 17 aligners: `Clustal` (Thompson et al., 1994), `Clustal2` (Larkin et al., 2007), `Clustal Omega` (Sievers et al., 2011), `DIALIGN` (Subramanian et al., 2008), `FSA` (Bradley et al., 2009), `Kalign` (Lassmann and Sonnhammer, 2005a), `MAFFT` (Katoh et al., 2005), `MUMMALS` (Pei and Grishin, 2006), `Muscle` (Edgar, 2004a), `MSAProbs` (Liu et al., 2010), `Opal` (Wheeler and Kececioglu, 2007), `POA` (Lee et al., 2002), `PRANK` (Loytynoja and Goldman, 2005), `PROBALIGN` (Roshan and Livesay, 2006), `ProbCons` (Do et al., 2005), `SATé` (Liu et al., 2011), and `T-Coffee` (Notredame et al., 2000).

7.2.2 Determining the universe of parameter settings

For *general aligner advising*, we selected a subset of the above aligners on which we enumerated values for their tunable parameters, to form a universe of parameter settings. We selected this subset of aligners by the following process. First, we computed an optimal oracle set of cardinality $k = 5$ over the universe of 17 aligners for default advising listed above. This set consisted of `Kalign`, `MUMMALS`, `Opal`, `PROBALIGN`, and `T-Coffee`. We then expanded this set further by adding four aligners that are used extensively in the literature: `Clustal Omega`, `MAFFT`, `Muscle`, and `ProbCons`. In the experiments described later in Section 7.3.2, we constructed greedy advisor sets over the universe of 17 aligners for default aligner advising, and noticed a large increase in advising accuracy at cardinality $[6, 8]$ (which can be seen in Figure 7.8). The greedy advisor sets at these cardinalities contained all of the aligners already chosen so far, with the addition of the `PRANK` aligner. Finally, we added `PRANK` to our set for this reason. The above 10 aligners comprise the set we considered for general aligner advising.

Table 7.1 lists the universe of parameter settings for these aligners for general advising. For each aligner, we enumerated parameter settings by forming a cross product of values for each of its tunable parameters. We determined the values for each tunable parameter by one of two ways. For aligners with web-server versions (namely `Clustal Omega` and `ProbCons`), we used all values recommended for each parameter. For all other aligners, we chose either one or two values above and below the default value for each parameter, to attain a cross product with less than 200 parameter settings. If a range was specified for a numeric parameter, values were chosen to cover this range as evenly as possible. For non-numeric parameters, we used all available options. Table 7.1 summarizes the resulting universe for general advising of over 800 parameter settings.

Table 7.1: ***Universe of Parameter Settings for General Aligner Advising***

Aligner	Parameter settings	Tunable parameters	Version	Parameter name	Default value, v	Alternate values
`Clustal Omega` (Sievers et al., 2011)	120[1]	5	1.2.0	Number of guide tree iterations	0	1, 3, 5
				Number of HMM iterations	0	1, 3, 5
				Number of combined iterations	0	1, 3, 5
				Distance matrix calculations, initial	mBed	Full alignments
				Distance matrix calculations, iterations	mBed	Full alignments
`Kalign` (Lassmann and Sonnhammer, 2005a)	162	4	2.04	Gap open penalty	55	40, 70
				Gap extension penalty	8.5	7, 10
				Terminal gap penalty	4.25	3.5, 5
				Bonus	No	Yes
`MAFFT` (Katoh et al., 2005)	100	3	6.923b	Substitution matrix	`BLSM62`	`BLSM80`, `VTML120`, `VTML200`
				Gap open penalty	1.53	$\frac{1}{4}v$, $\frac{1}{2}v$, $\frac{3}{2}v$, $2v$
				Gap extension penalty	0.123	$\frac{1}{2}v$, $2v$, $4v$
`Muscle` (Edgar, 2004a)	80	3	3.8.31	Profile score	Log-expectation: `VTML240`	Sum-of-pairs: `PAM200`, `VTML240`
				Objective function[2]	`spm`	`dp`, `ps`, `sp`, `spf`, `xp`
				Gap open penalty, profile dependent	v^3	$\frac{1}{2}v$, $\frac{3}{4}v$, $\frac{5}{4}v$, $\frac{3}{2}v$
`MUMMALS` (Pei and Grishin, 2006)	29	3[4]	1.01	Differentiate match states in unaligned regions	Yes	No
				Solvent accessibility categories	1	2, 3
				Secondary structure types	3	1
`Opal` (Wheeler and Kececioglu, 2007)	162	5	3.0b	Substitution matrix	`VTML200`[5]	`BLSM62`[5], `VTML40`[5]
				Internal gap open penalty	$\gamma = 45$	70, 95
				Terminal gap open penalty	0.4γ	0.05γ, 0.75γ
				Internal gap extension penalty	$\lambda = 42$	40, 45
				Terminal gap extension penalty	$\lambda - 3$	λ
`PRANK` (Loytynoja and Goldman, 2005)	50	3	.140603	Gap rate	0.005	$\frac{1}{5}v$, $\frac{1}{2}v$, $\frac{3}{2}v$, $2v$
				Gap extension	0.5	$\frac{1}{5}v$, $\frac{1}{2}v$, $\frac{3}{2}v$, $2v$
				Terminal gaps	Alternate scoring	Normal scoring
`PROBALIGN` (Roshan and Livesay, 2006)	64	3	1.4	Consistency repetitions	2	0, 1, 3
				Iterative refinement repetitions	100	0, 500
				Pre-training repetitions	0	1, 2, 3, 4, 5, 20
`ProbCons` (Do et al., 2005)	48[6]	3	1.12	Thermodynamic temperature	5	3, 5
				Gap open	22	11, 33
				Gap extension	1	0.5, 1.5
`T-Coffee` (Notredame et al., 2000)	36	3	10.00.r1613	Substitution matrix	`BLSM62`	`BLSM40`, `BLSM80`
				Gap open	0	-50, -500, -1000
				Gap extension	0	-5, -10
Total	856					

[1] Parameter settings retrieved from the `Clustal Omega` web-server at EBI (`www.ebi.ac.uk/Tools/msa/clustalo`).

[2] `sp`: sum-of-pairs score; `spf`: dimer approximation of sum-of-pairs score; `spm`: input dependent (`sp` if input is less than 100 sequences, `spf` otherwise); `dp`: dynamic programming score; `ps`: average profile sequence score; `xp`: cross profile score.

[3] Default values for the gap open penalty are -2.9 when the log-expectation profile is chosen, -1439 for sum-of-pairs using `PAM200`, and -300 for sum-of-pairs using `VTML240`. Alternate values are multiples of this default value.

[4] `MUMMALS` is distributed with 29 precomputed hidden Markov models, each of which is associated with a setting of three tunable parameters.

[5] The substitution matrices used by `Opal` are shifted, scaled, and rounded to integer values in the range $[0, 100]$.

[6] Parameter settings retrieved from the `ProbCons` web-server at Stanford (`probcons.stanford.edu`).

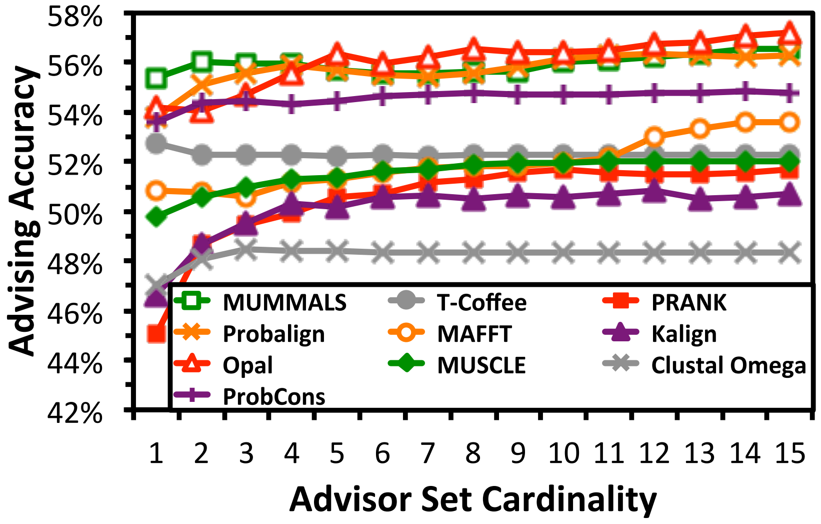


Figure 7.3: ***Accuracy of parameter advising using*** `Facet`. The plot shows advising accuracy for each aligner from Table 7.1, using parameter advising on greedy sets with the `Facet` estimator learned by difference fitting. The horizontal axis is the cardinality of the advisor set, and the vertical axis is the advising accuracy on testing data averaged over all benchmarks and folds, under 12-fold cross-validation.

7.3 Evaluating ensemble alignment

We evaluate the performance of advising through experiments on a collection of protein multiple sequence alignment benchmarks. A full description of the benchmark collection is given in Chapter 6, and is briefly summarized below. The experiments compare the accuracy of parameter and aligner advising to the accuracy of individual aligners using their default parameter settings.

The benchmark suites used in our experiments consist of reference alignments that are largely induced by performing structural alignment of the known three-dimensional structures of the proteins. Specifically, we use the `BENCH` suite of Edgar (2009), supplemented by a selection of benchmarks from the `PALI` suite (Balaji et al., 2001). The entire benchmark collection consists of 861 reference alignments.

As is common in benchmark suites, easy-to-align benchmarks are highly over-represented in this collection, compared to hard-to-align benchmarks. To correct for this bias when evaluating average advising accuracy, we binned the 861 benchmarks in our collection by *difficulty*, where the difficulty of a benchmark is its average accuracy under three commonly-used aligners, namely `Clustal Omega`, `MAFFT`, and `ProbCons`, using their default parameter settings. We then divided the full range $[0,1]$ of accuracies into 10 bins with difficulties $[(j-1)/10, j/10]$ for $j = 1, \ldots, 10$. The weight w_i of benchmark B_i falling in bin j that we used for training is $w_i = (1/10)(1/n_j)$, where n_j is the number of benchmarks in bin j. These weights w_i are such that each difficulty bin contributes equally to the advising objective function $f(P)$. Note that with this weighting, an aligner that on every benchmark gets an accuracy equal to its difficulty, will achieve an average advising accuracy of roughly 50%.

7.3.1 Parameter advising

We first examine the results of parameter advising for a single aligner using the `Facet` estimator. We learned the coefficients for `Facet` by difference fitting on computed alignments obtained using the oracle set of cardinality $k = 17$ found for the parameter universe for each aligner. (We trained the estimator on an oracle set of this cardinality to match the size of the universe for default aligner advising.) Given this estimator, we constructed greedy advisor sets for each aligner.

Figure 7.3 shows the accuracy of parameter advising using greedy advisor sets of cardinality $k \leq 15$, for each of the 10 aligners in Table 7.1, under 12-fold cross-validation. The plot shows advising accuracy on the testing data, averaged over all benchmarks and folds.

Almost all aligners benefit from parameter advising, though their advising accuracy eventually reaches a plateau. While our prior chapters showed that parameter advising boosts the accuracy of the `Opal` aligner, Figure 7.3 shows this result is not aligner dependent.

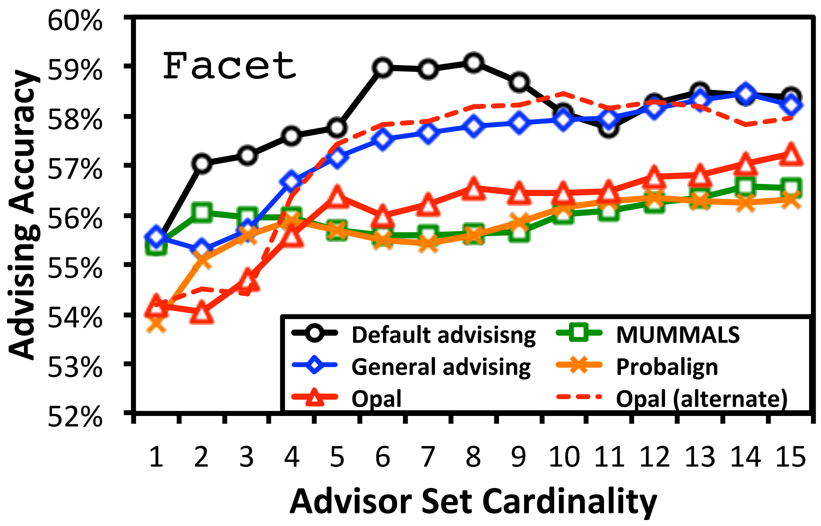


Figure 7.4: ***Aligner advising and parameter advising using*** `Facet`. The plot shows default and general aligner advising accuracy, and parameter advising accuracy for `Opal`, `MUMMALS`, and `PROBALIGN`, using the `Facet` estimator. The horizontal axis is the cardinality of the advisor set, and the vertical axis is advising accuracy on testing data averaged over all benchmarks and folds under 12-fold cross-validation.

7.3.2 Aligner advising

To evaluate aligner advising, we followed a similar approach, constructing an oracle set of cardinality $k = 17$ over the union of the universe for default advising from Section 7.2.1 and the universe for general advising from Section 7.2.2, learning coefficients for `Facet` using difference fitting, and constructing greedy sets using `Facet` for default and general advising.

Figure 7.4 shows the accuracy of default and general advising using greedy sets of cardinality $k \leq 15$, along with the three best parameter advising curves from Figure 7.3, for `Opal`, `PROBALIGN`, and `MUMMALS`. The plot shows advising accuracy on testing data, averaged over benchmarks and folds.

The dashed red curve in Figure 7.4 also shows the accuracy of `Opal` for parameter advising with greedy sets computed over an alternate universe of much more fine-grained parameter choices. This is the same universe used for the experiments in Chapter 6. Note that the dashed curve for parameter advising with `Opal`, using

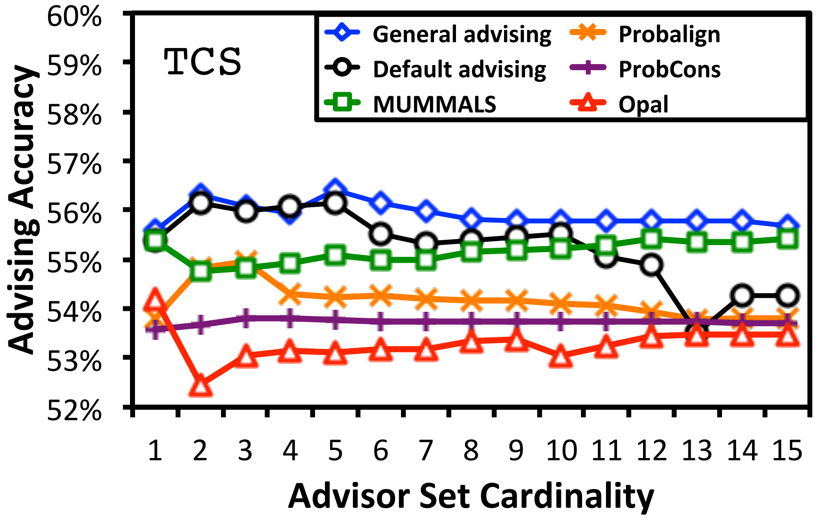


Figure 7.5: ***Aligner advising and parameter advising using*** `TCS`. The plot shows default and general aligner advising accuracy, and parameter advising accuracy for `Opal`, `MUMMALS`, `PROBALIGN`, and `ProbCons`, using the `TCS` estimator. The horizontal axis is the cardinality of the advisor set, and the vertical axis is advising accuracy on testing data averaged over all benchmarks and folds under 12-fold cross-validation.

greedy sets from these finer universes for each fold, essentially matches the accuracy of general advising at cardinality $k \geq 4$.

Testing the significance of improvement

To test the statistical significance of the improvement in default advising accuracy over using a single default aligner, we used the one-tailed Wilcoxon sign test (Wilcoxon, 1945). Performing this test in each difficulty bin, we found a significant improvement in accuracy ($p < 0.05$) on benchmarks with difficulty $(0.3, 0.4]$ at all cardinalities $2 \leq k \leq 15$, and on benchmarks with difficulty at most 0.4 at cardinality $6 \leq k \leq 9$.

We also tested the significance of the improvement of default advising over the best parameter advisor at each cardinality k (namely `MUMMALS` for $k \leq 4$ and `Opal` for $k \geq 5$), and found that at cardinality $k \geq 5$ there is again significant improvement ($p < 0.05$) on benchmarks with difficulty $(0.3, 0.4]$.

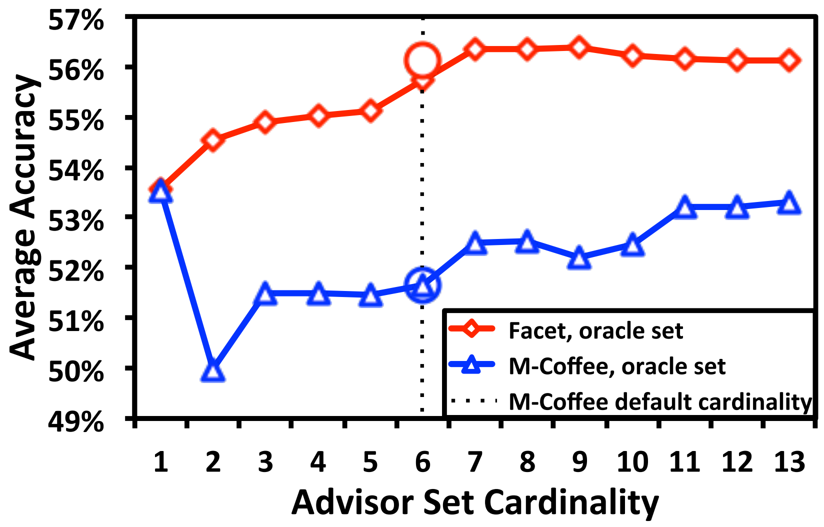


Figure 7.6: ***Accuracy of aligner advising compared to*** `M-Coffee`. The plot shows average accuracy for aligner advising using `Facet`, and meta-alignment using `M-Coffee`, on oracle sets of aligners. Performance on the default `M-Coffee` set of six aligners is indicated by large circles on the dotted vertical line. The horizontal axis is cardinality of the oracle sets, and the vertical axis is average accuracy on testing data over all benchmarks and folds under 12-fold cross-validation.

Advising with an alternate estimator

We also evaluated in the same way parameter advising and aligner advising on greedy sets using the `TCS` estimator (Chang et al., 2014) (the best other estimator for advising from the literature). Figure 7.5 shows results using `TCS` for parameter advising (on the four most accurate aligners), and for general and default aligner advising. Note that while `TCS` is sometimes able to increase accuracy above using a single default parameter, this increase is smaller than for `Facet`; moreover, `TCS` often has a decreasing trend in accuracy for increasing cardinality.

7.3.3 Comparing ensemble alignment to meta-alignment

Another approach to combining aligners is the so-called *meta-alignment* approach of `M-Coffee` (Wallace et al., 2006) (described in Section 7.1.1). `M-Coffee` computes a multiple alignment using position-dependent substitution scores obtained from alternate alignments generated by a collection of aligners. By default, `M-Coffee` uses the

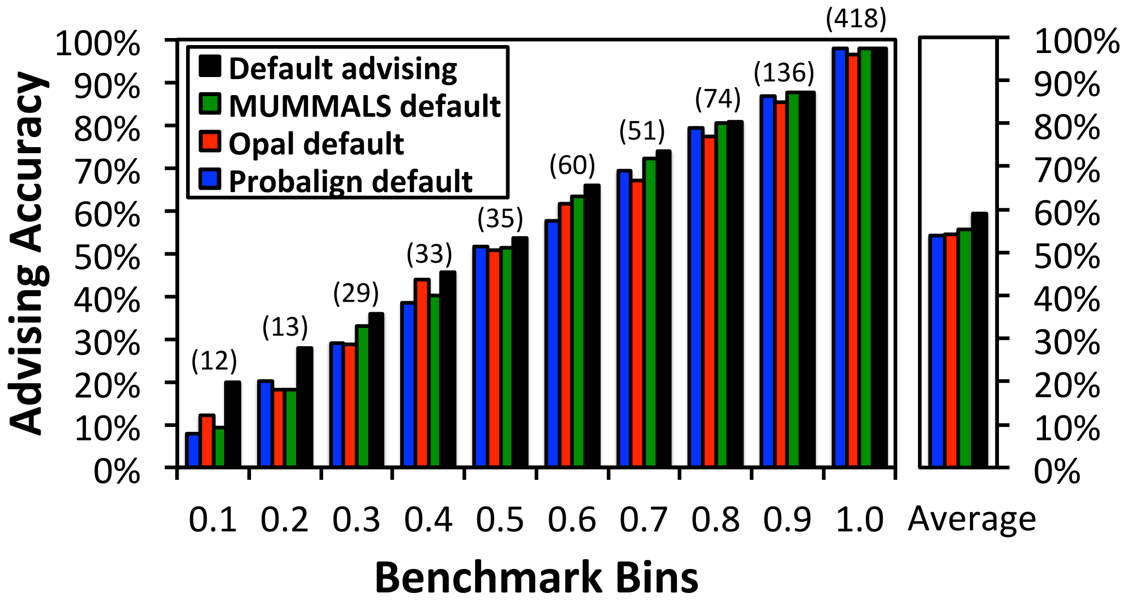


Figure 7.7: ***Accuracy of default aligner advising, and aligners with their default settings, within difficulty bins.*** In the bar chart on the left, the horizontal axis shows all ten benchmark bins, and the vertical bars show accuracy averaged over just the benchmarks in each bin. The accuracy of default advising using the `Facet` estimator is shown for the greedy sets of cardinality $k = 5$, along with the accuracy of the default settings for `PROBALIGN`, `Opal`, and `MUMMALS`. The bar chart on the right shows accuracy uniformly averaged over the bins. In parentheses above the bars are the number of benchmarks in each bin.

following six aligners: `Clustal2`, `T-Coffee`, `POA`, `Muscle`, `MAFFT`, `Dialign-T` (Subramanian et al., 2005), `PCMA` (Pei et al., 2003), and `ProbCons`. The tool also allows use of `Clustal`, `Clustal Omega`, `Kalign`, `AMAP` (S. Schwartz and Pachter, 2007), and `Dialign-TX`. Figure 7.6 shows the average accuracy of both `M-Coffee` and our ensemble approach with `Facet`, using the default aligner set of `M-Coffee` (the dotted vertical line with large circles), as well as oracle sets constructed over this `M-Coffee` universe of 13 aligners. Notice that at all cardinalities our ensemble aligner substantially outperforms meta-alignment even on the subset of aligners recommended by the `M-Coffee` developers.

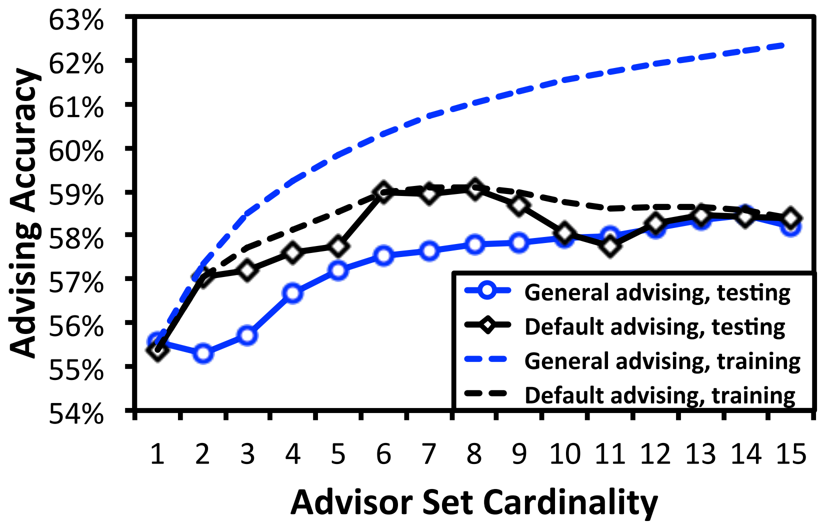


Figure 7.8: ***General and default aligner advising on training and testing data***. The plot shows general and default aligner advising accuracy using `Facet`. Accuracy on the training data is shown with dashed lines, and on the testing data with solid lines. The horizontal axis is cardinality of the advisor set, and the vertical axis is advising accuracy averaged over all benchmarks and folds under 12-fold cross-validation.

7.3.4 Advising accuracy within difficulty bins

Figure 7.7 shows advising accuracy within difficulty bins for default aligner advising compared to using the default parameter settings for the three aligners with highest average accuracy, namely `MUMMALS`, `Opal`, and `PROBALIGN`. The figure displays the default advising result from Section 7.3.2 at cardinality $k = 5$. The bars in the chart show average accuracy over the benchmarks in each difficulty bin, as well as the average accuracy across all bins. (The number of benchmarks in each bin is in parentheses above the bars.) Note that aligner advising gives the greatest boost for the hardest-to-align benchmarks: for the bottom two bins, advising yields an 8% increase in accuracy over the best aligner using its default parameter setting.

7.3.5 Generalization of aligner advising

The results thus far have shown advising accuracy averaged over the testing data associated with each fold. We now compare the training and testing advising accuracy to assess how our method might generalize to data not in our benchmark set.

Figure 7.8 shows the average accuracy of default and general aligner advising on both training and testing data. Note that the drop between training and testing accuracy is much larger for general advising than for default advising, resulting in general advising performing worse than default advising though its training accuracy is much higher. This indicates that general advising is strongly overfitting to the training data, but could potentially achieve much higher testing accuracy. Additionally, there is a drop in training accuracy for default advising with increasing cardinality, though after its peak an advisor using greedy sets should remain flat in training accuracy as cardinality increases when using a strong estimator.

7.3.6 Theoretical limit on advising accuracy

An ***oracle*** is an advisor that uses a perfect estimator, always choosing the alignment from a set that has highest true accuracy. To examine the theoretical limit on how well aligner advising can perform, we compare the accuracy of aligner advising using `Facet` with the performance of an oracle. Figure 7.9 shows the accuracy of both default and general aligner advising using greedy sets, as well as the performance of an oracle using oracle sets computed on the default and general advising universes. (Recall an *oracle set* is an optimal advisor set for an oracle.) The plot shows advising accuracy on testing data, averaged over all benchmarks and folds. The large gap in performance between the oracle and an advisor using `Facet` shows the increase in accuracy that could potentially be achieved by developing an improved estimator.

7.3.7 Composition of advisor sets

Table 7.2 lists the greedy advisor sets for both default and general advising for all cardinalities $k \leq 10$. A consequence of the construction of greedy advisor sets is that the greedy set of cardinality k consists of the entries in a column in the first k rows of the table. The table shows these sets for just one fold from the 12-fold

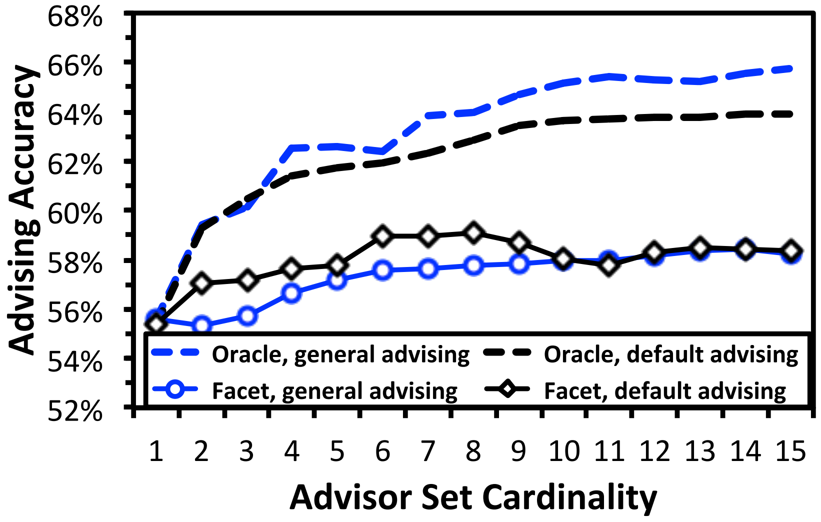


Figure 7.9: ***Accuracy of aligner advising using a perfect estimator.*** The plot shows advising accuracy for default and general aligner advising, both on oracle sets for a perfect estimator, and on greedy sets for the `Facet` estimator. The horizontal axis is the cardinality of the advisor set, and the vertical axis is advising accuracy on testing data averaged over all benchmarks and folds under 12-fold cross-validation.

cross-validation. For general advising sets, an entry specifies the aligner that is used, and for aligners from the general advising universe, a tuple of parameter values in the order listed in Table 7.1. The two exceptions are `MUMMALS`, whose 6-tuple comes from its predefined settings file, and whose last three elements correspond to the three parameters listed in Table 7.1; and `MSAProbs`, whose empty tuple stands for its default setting. It is interesting that other than `MSAProbs`, the general advising set does not contain any aligner's default parameter settings, though its values are close to the default setting.

7.3.8 Running time for advising

We compared the time to evaluate the `Facet` estimator on an alignment to the time needed to compute that alignment by the three aligners used for determining alignment difficulty: `Clustal Omega`, `MAFFT`, and `ProbCons`. To compute the average running time for these aligners on a benchmark, we measured the total time for

Table 7.2: ***Greedy Default and General Advising Sets***

	Default advising	General advising
1	`MUMMALS`	`MUMMALS` (0.2, 0.4, 0.6, 1, 2, 3)
2	`Opal`	`Opal` (VTML200, 45, 2, 45, 45)
3	`PROBALIGN`	`Opal` (BLSM62, 70, 3, 45, 42)
4	`Kalign`	`MUMMALS` (0.15, 0.2, 0.6, 1, 1, 3)
5	`Muscle`	`Opal` (BLSM62, 45, 33, 42, 42)
6	`T-Coffee`	`MSAProbs` ()
7	`PRANK`	`Kalign` (55, 8.5, 4.25, 0)
8	`Clustal Omega`	`MAFFT` (VTML200, 0.7515, 0.492)
9	`DIALIGN`	`Opal` (BLSM62, 95, 4, 45, 42)
10	`ProbCons`	`Opal` (BLSM62, 45, 2, 45, 42)

each of these aligners to align all 861 benchmarks on a desktop computer with a 2.4 GHz Intel i7 8-core processor and 8 Gb of RAM. The average running time for `Clustal Omega`, `MAFFT`, and `ProbCons` was less than 1 second per benchmark, as was the average running time for `Facet`. As stated in Chapter 3 the time complexity for `Facet` is dependent on the number of columns in an alignment, and should take relatively less time than computing an alignment for benchmarks with long sequences; the standard benchmark suites tend to include short sequences, however, which are fast to align. This time to evaluate `Facet` does not include the time to predict protein secondary structure, which is done once for the sequences in a benchmark, and was performed using `PSIPRED` (Jones, 1999) version `3.2` with its standard settings. Secondary structure prediction with a tool like `PSIPRED` has a considerably longer running time than alignment, due to an internal `PSI-BLAST` search during prediction; on average, `PSIPRED` took just under 6 minutes per benchmark to predict secondary structure.

7.4 Software

Our new ensemble aligner is implemented using `Perl` as a wrapper around the various underlying aligners. The `Perl` programs can be used in one of two ways:

(1) ***Using the predefined set program*** – The `Facet` release

comes with two applications, `default_ensemble_alignment.pl` and `ensemble_alignment.pl`, which can be used to run default and general ensemble alignment on sets learned from all training benchmarks. To use these applications, you provide the set cardinality you would like to use, the unaligned sequences, and predicted secondary structure. The application then runs each program in order, and outputs the result to standard out.

(2) ***Using a program that accepts an advisor set*** – We have also included an application `ensemble_alignment_from_set.pl` that accepts the input unaligned sequences, secondary structure prediction, and an advisor set similar to the one defined in earlier chapters for parameter advising. The advisor set contains the aligner and parameter setting information that will be used to run the aligners. Each line of the file contains one aligner and parameter setting in the format $\mathcal{A}\_\mathcal{S}$ where $\mathcal{A}$ is the aligner name and $\mathcal{S}$ is the tuple of parameter settings for that aligner separated by "." characters; for example, the default `Opal` parameter setting would be `Opal_VTML`200.45.11.42.41. Parameter advising sets for each of the applications tested, as well as default and general advising parameter sets in the proper format, can be found on the `Facet` website.

For both of these methods, the applications must be edited if any of the applications being used are not in the default installation location.

Summary

In this work, we have extended parameter advising to *aligner advising*, to yield a true *ensemble aligner*. Parameter advising gives a substantial boost in accuracy for nearly all aligners currently used in practice. Furthermore, default and general aligner advising both yield further boosts in accuracy, with default advising having better generalization. As these results indicate, ensemble alignment by aligner advising is a promising approach for exploiting future advances in aligner technology.

CHAPTER 8

Adaptive Local Realignment

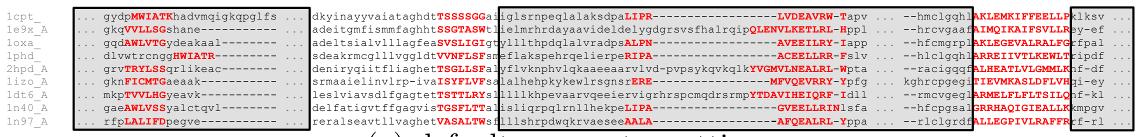
Overview

Mutation rates differ across the length of most proteins, but when multiple sequence alignments are constructed for protein sequences, a single alignment parameter choice is often used across the entire length. We provide for the first time an approach called ***adaptive local realignment*** that computes protein multiple sequence alignments using diverse parameter settings for different regions of the input sequences. In this way, parameter choices can vary across the length of a protein to more closely model the local mutation rate.

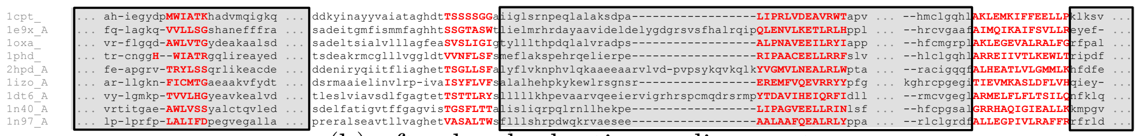
Using adaptive local realignment boosts alignment accuracy over using a default parameter choice. In addition, when adaptive local realignment is combined with global parameter advising, we see an increase in accuracy of almost 24% over the default parameter choice on hard-to-align benchmarks.

8.1 Introduction

Since the 1960s it has been known that proteins can have distinct mutation rates at different locations along the molecule (Fitch and Margoliash, 1967). The amino acids at some positions in a protein may stay unmutated for long periods of time, while other regions change a great deal (often called "hypermutable" regions). This has led to methods in phylogeny construction that take variable mutation rates into account when building trees from sequences (Yang, 1993). In multiple sequence alignment, however, to our knowledge variation in mutation rates across sequences has yet to be exploited to improve alignment accuracy. Multiple sequence alignments are typically computed using a single setting of values for the parameters of the



(a) default parameter settings



(b) after local adaptive realignment

Figure 8.1: Impact of adaptive local realignment. The figure shows portions of an alignment of benchmark `BB11007` from the `BAliBASE` suite, where the highlighted amino acids in red uppercase are from the core columns of the reference alignment, which should be aligned in a correct alignment. (a) The alignment computed by `Opal` using its optimal default parameter setting (`VTML200`, 45, 11, 42, 40) across the sequences, with an accuracy of 89.6%. The regions of the alignment in gray boxes are automatically selected for realignment. (b) The outcome of using adaptive local realignment, with an improved accuracy of 99.6%. The realignments of the three regions use alternate parameter settings (`BLOSUM62`, 45, 2, 45, 42), (`BLOSUM62`, 95, 38, 40, 40), and (`VTML200`, 45, 18, 45, 45), respectively, which increase the accuracy of these regions.

alignment scoring function. This single parameter setting affects how residues across a protein are aligned, and implicitly assumes a uniform mutation rate. In contrast, the approach of this paper identifies alignment regions that may be misaligned under a single parameter setting, and finds alternate parameter settings that may more closely match the local mutation rate of the sequences.

We present a method that takes a given alignment and attempts to improve its overall accuracy by replacing sections of it with better subalignments, as demonstrated in Figure 8.1. The top alignment of the figure was computed using a single parameter setting: the optimal default setting of the `Opal` aligner (Wheeler and Kececioglu, 2007). The bottom alignment is obtained by our new method, taking the top alignment, automatically identifying the sections in gray boxes, and realigning them using alternate parameter settings, as described later in Section 8.2. This increases the overall alignment accuracy by 10%, as most of the misaligned core blocks (highlighted in red uppercase) are now corrected.

Related work

Methods that partition a set of sequences to align or realign them can be divided in two categories, based on the type of partition. *Vertical* realigners cut the input sequences into substrings, and once these shorter substrings are realigned, they stitch their alignments together. *Horizontal* realigners split an alignment into groups of whole sequences, which are then merged together by realigning between groups, possibly using each group's induced subalignment.

`Crumble` and `Prune` (Roskin et al., 2011) is a pair of algorithms for performing both vertical (`Crumble`) and horizontal (`Prune`) splits on an input set of sequences. During the `Crumble` stage, a set of constraints is found that anchor the input sequences together, and the substrings or blocks between these anchor points are aligned. Once the disjoint blocks of the sequences are aligned, they are then fused by aligning their overlapping anchor regions. During the `Prune` stage, smaller groups of sequences are aligned that correspond to subtrees of the progressive aligner's guide tree. The subset of sequences in a subtree is then replaced by their alignment's consensus sequence in the remaining steps of progressive alignment. The original subalignments of the groups are finally reinserted to form the output alignment. Replacing a group of sequences by their consensus sequence during alignment reduces the number of sequences that are aligned at any one time. The objective for splitting sequences both vertically and horizontally within `Crumble` and `Prune` is to reduce time and space usage to make feasible the alignment of large numbers of long sequences, rather than to improve alignment accuracy.

`ReAligner` (Anson and Myers, 1997) is a horizontal realignment method for improving DNA sequence assembly by removing and then realigning sequencing reads. If a read is initially misaligned in the assembly it may be corrected when the read is removed and realigned. This realignment process is repeated over all reads to continually refine the assembly.

Gotoh (1993) presented several horizontal methods for heuristically aligning two multiple sequence alignments, which he called "group-to-group" alignment. This could be used for alignment construction in a progressive aligner, proceeding bottom-up over the guide tree and applying group-to-group alignment at each node, or for polishing an existing alignment by assigning sequences to two groups and using it to realign the groups.

`AlignAlign` (Kececioglu and Starrett, 2004) is a horizontal method that implements an exact algorithm for optimally aligning two multiple sequence alignments under the sum-of-pairs scoring function with affine gap costs. This optimal group-to-group alignment algorithm, used for both alignment construction and alignment polishing, forms the basis of the `Opal` aligner (Wheeler and Kececioglu, 2007).

By comparison, adaptive local realignment is a vertical approach that aims to improve alignment accuracy, applies to any alignment method that has tunable parameters, and to our knowledge is the first approach to alignment that can automatically adapt to varying mutation rates along a protein.

Plan of the chapter

In the next section, we describe our adaptive local realignment method, which can be viewed as a form of local parameter advising, and discusses how it interacts with global parameter advising. Section 8.3 experimentally evaluates our approach, and compares it with prior methods for advising.

8.2 Adaptive local realignment

To overcome the issue of protein sequences being non-homogeneous and having regions that may require different alignment parameters we have developed a method we call *adaptive local realignment.* Adaptive local realignment uses some of the the same basic principles that have been shown to work well for global parameter advising. We apply the techniques described above locally to choose the best alignment parameters for a subset of columns of an alignment.

The adaptive local realignment method for an alignment can be broken down into two steps: (1) choosing regions of the alignment that are correctly aligned which we should save, and (2) producing a new alignment for those regions that are not correctly aligned.

Similar to global parameter advising local realignment relies on a set of alternate parameter choices and the accuracy estimator.

8.2.1 Identifying local realignment regions

Just as with global alignments we do not have a known reference alignment so we cannot simply identify the alignment columns that are recovered correctly in an input computed alignment. Therefore we are forced to use an accuracy estimator E to define the regions of a given alignment that are going to be saved (and which will be realigned). We calculate the estimated accuracy of a sliding window across the alignment (see Figure 8.2a). The window size is a fraction $w \leq 1$ of the total length of the alignment. The window size w must be chosen carefully because the accuracy estimator has features that are global calculations of an alignment. A larger sliding window will provide more context at each position and should provide a better estimate of accuracy. At the same time, if the window is too large there will not be enough granularity to identify the separation points between correct and incorrectly aligned columns. To account for very short and very long alignments a minimum w_{min} and maximum w_{max} window size is specified.

We now have the scores for approximately $\frac{1}{w}$ windows that overlap each column of an alignment. We calculate a score for each column as a sum of these scores weighted proportionally to the distance to the center column of that window (see Figure 8.2b). We use a gamma decay distribution with a decay factor $d \leq 1$ centered on the middle column to weight the contribution of each window. As d approaches 1 a column gets equal weight from all covering windows. Conversely as it approaches 0 the score is dependent only on the window centered at that column.

We then calculate two thresholds τ_B and τ_S based on the percentage of columns from the original alignment we would like to keep T_B and the percentage of columns we will use to seed realignment regions T_S. The thresholds are set so that at least $\lceil \ell \, T_B \rceil$ columns have score that are above τ_B, and least $\lceil \ell \, T_S \rceil$ columns have scores

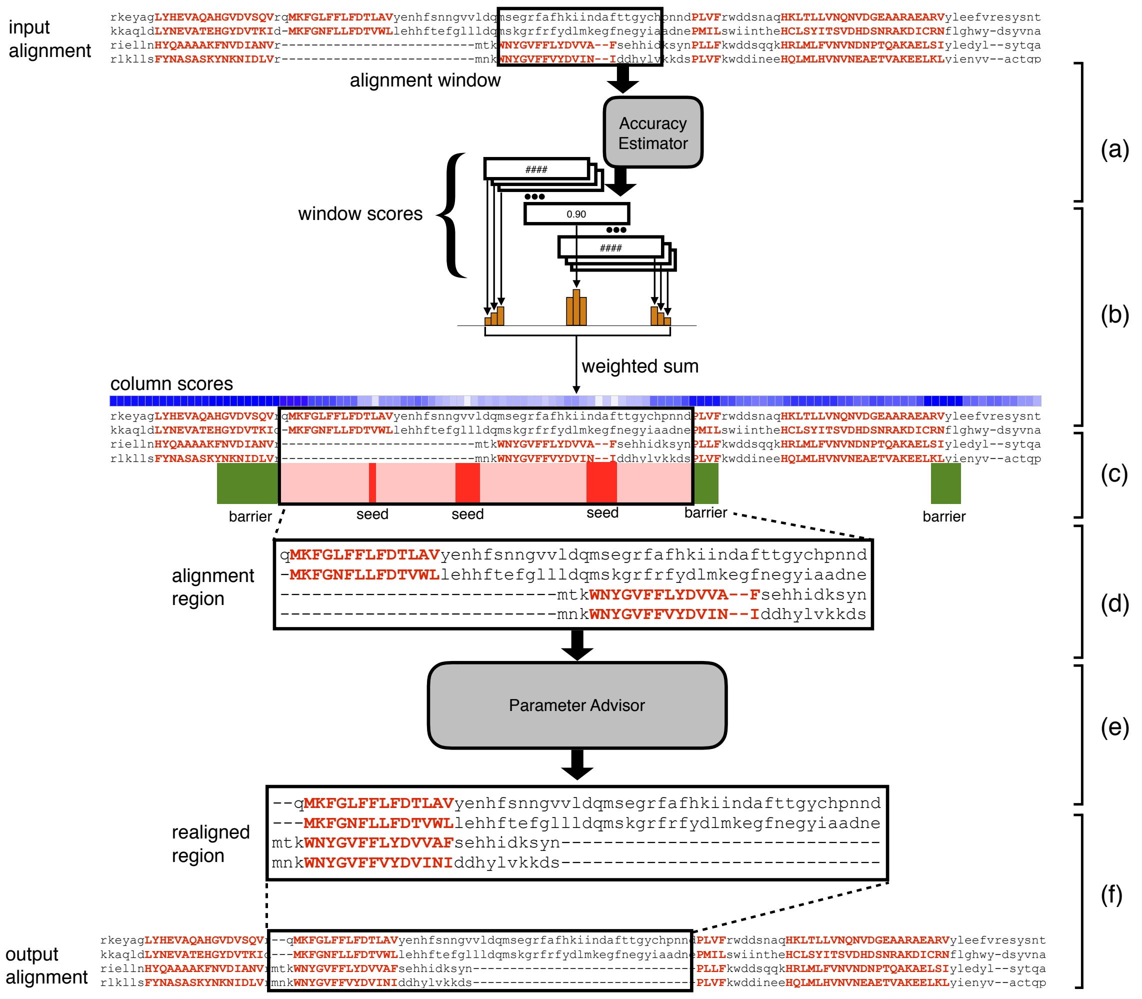


Figure 8.2: The adaptive local realignment process. (a) We calculate a Facet score for a sliding window across at the input alignment. (b) To calculate a score for each column from the set of window scores we use a weighted sum of the values for all windows that overlap that column. (c) Columns that a column score value greater than τ_G are labeled as barriers and then columns with value less than τ_B are used as seeds for realignment regions. (d) These seeds are then extended in both directions until they reach a barrier column to define a realignment region that is extracted from the alignment. (e) The unaligned subsequences defined by this region are then realigned using a parameter advisor. (f) Once the most accurate realignment of the region is found it is reinserted into the input alignment replacing the section that was removed.

below τ_S. All columns with scores $\geq \tau_B$ are labeled “barriers” and all columns with scores $\leq \tau_S$ are labeled as “seeds” (see Figure 8.2b).

To find alignment regions that will be realigned we start a region by including a seed column. This region is then extended to include any other seed or unlabeled column to the left and right. This expansion continues until a barrier column (or the end of the alignment) is reached in both directions.

The barrier columns will never be included in an alignment region that will be realigned. In this way we guarantee that at least T_B percent of the columns from the original alignment will remain, and there will always be at least one alignment region to realign.

8.2.2 Local parameter advising on a region

During local advising we will construct a new alignment that contains all of the columns that are surrounded by only barrier columns and more accurate alignments of the columns covered by alignment regions (if a more accurate alignment can be found).

For each alignment region we extract the sub-alignment in the contained columns and calculate it's `Facet` score (see Figure 8.2d). We will later compare other alternate alignments with this base accuracy. The unaligned subsequences of this region are then collected. This set of unaligned sequences becomes the input to parameter advising.

We use same parameter advising method described in Section 1.2 and Figure 1.3 with one exception. The `Opal` aligner has 5 tunable parameters: the replacement matrix as well as two internal and two terminal gap costs (we describe them in detail in Section 8.3). For those regions that do include the alignment terminals (the first or last column of the input alignment), the terminal gap penalties are replaced with the corresponding internal gap cost. For those regions that do include terminals we use the terminal gap penalty only on the one side that is terminal in the context of the global alignment. Note that an alignment region as we have defined it will never include both terminals.

We then compare the advisors choice with the original alignment of this region, if the accuracy of the new alignment is higher we remove the columns covered by the alignment region from the input alignment and replace them with the new alignment of this region (see Figure 8.2f).

As a final step we compare the accuracy of the new alignment, with the realignments in the alignment regions, to the input alignment. The more accurate global alignment is returned.

8.2.3 Iterative local realignment

Once we have a new global alignment of the input sequences we can repeat the process to continue to refine the alignment. Using the same methods described earlier we compute the `Facet` score on windows of this new alignment, combine these to get column scores, and define alignment regions for which we will use parameter advising to realign. We iterate this process until a user defined maximum number of iterations is reached. Note that the local advising procedure may reach a point where none of the misaligned regions are replaced, even though continuing iteration will not effect the output we stop iterating when this happens to reduce the running time.

8.2.4 Combining local with global advising

Local advising is a method for improving the accuracy of an existing alignment. It has been shown previously that using parameter choices other than the default can greatly increase the alignment accuracy for some inputs. We can use global advising to find a more accurate starting point for adaptive local realignment.

Local and global advising can be combined in two ways.

(1) **Local advising on *all* global alignments:** using adaptive local re-alignment on each of the alternate alignments produced within global parameter advising then choosing among all $2|P|$ alternate alignments ($|P|$ unaltered global alignments and $|P|$ locally advised alignments),

and

(2) **Local advising on *best* global alignment:** choosing the best global alignment then using adaptive local realignment to boost it's accuracy.

We will compare both methods for combining local and global advising as well as local advising on the default alignment in the next section.

8.3 Assessing local realignment

We evaluate the performance of adaptive local realignment and its use in combination with global advising through experiments on a collection of protein multiple sequence alignment benchmarks. A full description of the benchmarks and universe of parameters used for parameter advising can be found in Kececioglu and DeBlasio (2013) and is briefly described here.

The benchmark suites used in our experiments consist of reference alignments of proteins that are largely induced by structurally aligning their known three-dimensional structure. In particular, we use the `BENCH` suite of Edgar (2009) (which is a combination of the `BAliBASE` (Bahr et al., 2001), `PREFAB` (Edgar, 2004a), `OxBench` (Raghava et al., 2003), and `SABRE` (Van Walle et al., 2005) databases), supplemented by a selection from the `PALI` suite of Balaji et al. (2001). The full benchmark collection we use consists of 861 reference alignments.

As is common in benchmark suites, easy-to-align benchmarks are highly over-represented in this collection. To correct for this bias towards easy to align benchmarks when evaluating average advising accuracy, we binned the 861 benchmarks by *hardness*, which we measured by the true accuracy of the alignment of the benchmark's sequences using the multiple alignment tool `Opal` under the optimal default parameter setting. We then divided the the full range $[0, 1]$ of accuracies into 10 bins, where bin b for $b = 1, ..., 10$ contains hardness interval $\big((b-1)/10, b/10\big]$, and has 12, 12, 20, 34, 26, 50, 62, 74, 137, and 434 benchmarks respectively. We report the average accuracy across *bins* rather than across benchmarks. This means that the average accuracy of alignments using the `Opal` default parameter settings is near 50%. Even though the binning is based on the Opal default alignments, most other standard aligners have default accuracy near 50%: Clustal Omega (Sievers et al., 2011, 47.3%), Muscle (Edgar, 2004a, 48.4%), MAFFT (Katoh et al., 2005, 51.0%). This is not to say for instance that MAFFT is necessarily more accurate than Clustal Omega, if you bin based on any aligner other than Opal you would get a completely new ordering. DeBlasio and Kececioglu (2016) shows that for the task of parameter advising many of the top aligners performance nearly equally well and our choice to use Opal is made based on the fact that it has the highest advising accuracy in our tests. The methodology presented here is general and can be implemented for any other aligner.

We developed a universe of alignment parameter settings U by enumerating the tunable alignment parameters within the Opal aligner and enumerated values from within the reasonable range of those parameters. In particular the tunable parameters for Opal are represented as a 5-tuple $(\sigma, \lambda_I, \lambda_T, \gamma_I, \gamma_T)$ which represent the replacement matrix (σ) as well as the the internal and terminal gap open (λ) and extension costs (γ). For the substitution matrix we selected 3 matrices form the BLOSUM (Henikoff and Henikoff, 1992) and VTML (Müller et al., 2002) families, three choices each for the internal and external gap open costs, three choices of internal gap extension cost, and two choices of terminal gap extension costs. We then took the cross product of the choices for each of the parameters to generate a universe of 162 parameter settings.

We use *12-fold cross validation* to examine the increase in accuracy gained using local advising both with and without the addition of global advising. We construct training and testing subsets of the alignment benchmarks by evenly and randomly distributed benchmarks into twelve groups for each hardness bin; we then formed twelve splits of the entire collection of benchmarks into a training class and a testing class, where each split placed one group in a bin into the training class and the other eleven groups in the bin into the training class; finally, for each split we generated a *training set* and *testing set* of examples alignments as follows: for each benchmark B in a training or testing class, we generate $|U|$ example alignments in the respective training or testing set by running `Opal` on B with each parameter in U. An estimator learned on the examples from a training set was evaluated on examples from the corresponding testing set. The results we reported are averages over twelve folds, where each *fold* is one of these pairs of associated training and testing sets. (Note that across twelve folds, every example is tested on exactly once.)

We trained the estimator coefficients for `Facet` using the difference fitting method described in Chapter 2 on the training sets described above. We found that there was very little change in coefficients between the training folds so for ease of experimentation we use the estimator coefficients that are release with the newest version `Facet` which were trained on all available benchmarks.

We examined several settings for the tunable parameters of the local realignment method: estimator window size percentage w (10%,20%,30%,...,90%), minimum window sizes w_{min} (5,10,20,30), minimum window sizes w_{max} (30,50,75,100,125), good and bad column label percentages B_G, B_B (5%,10%,20%,30%,...,70%), and gamma decay value d (0.5,0.66,0.9,0.99). We used the performance on *training* benchmarks described above to find the combination of these settings that gave the highest improvement in accuracy when local advising was applied to the default alignments from `Opal`. We found that using $w = 30\%$, $w_{min} = 10$, $w_{max} = 30$, $B_G = 10\%$, $B_B = 30\%$, and $d = 0.9$ provided the highest increase for the training benchmarks and these are the settings we use through out the experimental results. We also iterate the local advising step five times and use this for all experiments other than the comparison to `TCS` in Section 8.3.3, full details why we used five iterations are shown in Section 8.3.4.

8.3.1 Effect of local realignment across difficulty bins

Figure 8.3 shows the alignment accuracy across difficulty bins for default alignments from `Opal`, local advising on these default alignments, global advising alone, and local combined with global alignment. Here the combination method uses local advising on all alternate alignments within global advising. The oracle set of cardinality $k = 10$ was used for both global and local advising.

Local advising greatly improves the alignment accuracy of default alignments (left two bars in each group). In the two most difficult benchmark bins (to the left of the figure) using local advising increases the average accuracy by 11.5% and 9.1% respectively. The accuracy increases on all bins. Overall using local advising increases the accuracy of the default alignments by and average of 4.5% across bins.

Combining local and global advising greatly improves the accuracy over either of the methods individually. This is most pronounced for the hardest to align benchmarks. For the bottom two bins using both parameter advising and adaptive local realignment increases the accuracy by 23.0% and 25.6% accuracy over using just the default parameter choices. Additionally, using adaptive local realignment increases the accuracy by 5.9% accuracy on the bottom most bins over using parameter advising alone. On average thats an 8.9% increase in accuracy over all bins by using the combined procedure over using just the default parameter choice and a 3.1% increase over using only parameter advising.

8.3.2 Varying advising set cardinality

In previous sections we focused on using advising sets of cardinality $k = 10$. Because an alignment is produced for each region of local realignment for each parameter choice in the advising set it may be desirable to use a smaller set to reduce the running time of local (or or global) advising. We produced oracle advising sets for cardinalities $k = 2...15$ and used them to test the effect of local advising both alone and in combination with global advising. Figure 8.4 shows the average advising accuracies of using advising sets of increasing cardinalities under the 3 conditions described above as well as the combination of local with global advising where the local advising step is only performed on the single best alignment identified by global advising. The cardinality of the set used for both parameter advising and local realignment is shown on the horizontal axis, while the vertical axis shows the alignment accuracy of the produced alignments averaged first within difficulty bins then across bins.

The accuracy of alignments produced by all four methods shown eventually

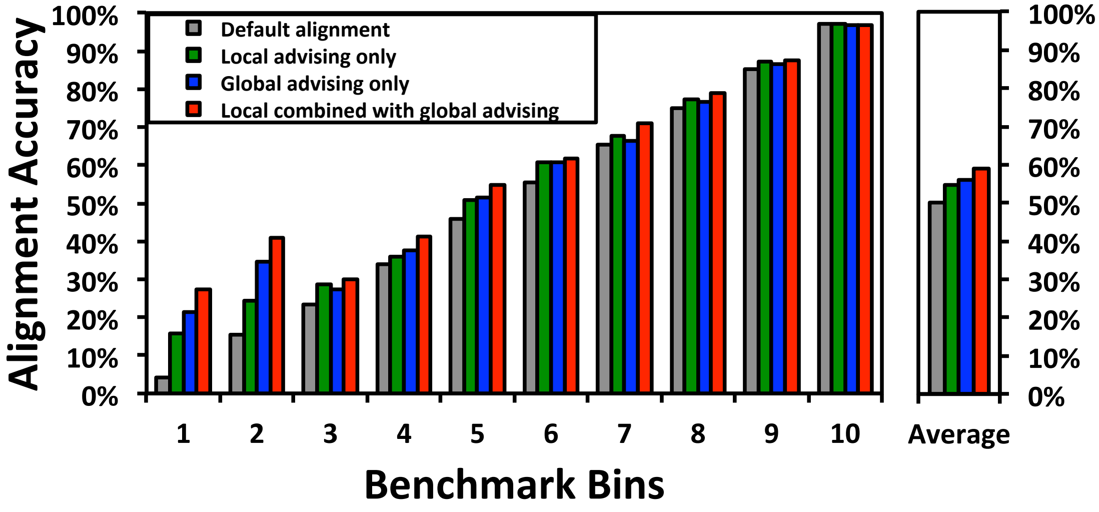


Figure 8.3: ***Accuracy of the default alignment, local realignment of the default alignment, parameter advising, and parameter advising with local realignment within difficulty bins.*** In the bar chart on the left the horizontal axis shows all ten benchmarks bins, and the vertical bars show the accuracy averaged over just the benchmarks in that bin. The accuracy of the default alignment and parameter advising using an oracle set of cardinality $k = 10$, before local realignment is shown as well as the application of local realignment to both results. The car chart on the right shows the accuracy uniformly averaged over the bins.

reaches a plateau where adding additional parameters to the advising set no longer increases the alignment accuracy. This plateau is reached at cardinality $k = 10$ when local realignment is applied to the default alignments and at $k = 6$ for parameter advising with and without local realignment, but this plateau is higher for the combined methods.

Across all cardinalities using local combined with global advising improves alignment accuracy by nearly 4% on average.

The results above give average advising accuracy uniformly weighted across *bins*. We now report average advising accuracy uniformly weighted across *benchmarks*. Using its default parameter choice the `Opal` aligner achieves accuracy 80.4%. Applying both local and global advising at cardinality $k = 10$, this increases to 83.1% (performing local advising on all global alternate alignments). Using only local or global advising achieves accuracy 82.1% or 81.8% respectively. At $k = 5$ the accuracy of using local and global advising is 82.7%. By comparison, the average accuracy of other standard aligners on these benchmarks is: `Clustal Omega`, 77.3%; `Muscle`, 78.09%; `MAFFT`, 79.38%.

8.3.3 Comparing estimators for local advising

We have shown previously that the `Facet` estimator has the best performance for the task of global advising compared to the other accuracy estimators available (see Kececioglu and DeBlasio, 2013; DeBlasio and Kececioglu, 2016). Figure 8.5 shows the average accuracy of local advising on default alignments using both `Facet` and `TCS` (the next best estimator for advising) using advisor sets of cardinality $k = 2...15$. We found that using `TCS` for local advising greatly increased the running time because it is an additional system call and additional file IO. Because of the additional computational requirements we did not iterate the local advising for either estimator. Using `TCS` for local advising gives an increase in accuracy of less than half that of `Facet`.

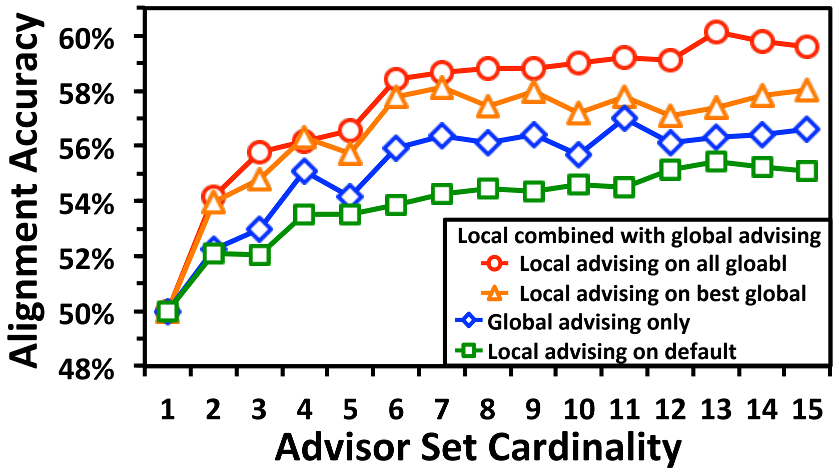


Figure 8.4: Advising accuracy using various methods versus set cardinality. This figure compares the accuracy of alignments produced by local advising on the alignment produced using the `Opal` default parameter settings, global advising alone, and two variants on combining local and global advising. The horizontal axis represents and increasing oracle advising set cardinality used for both parameter advising and local realignment. The vertical axis shows the accuracy of the alignments produced by each of the advising methods averaged across difficulty bins.

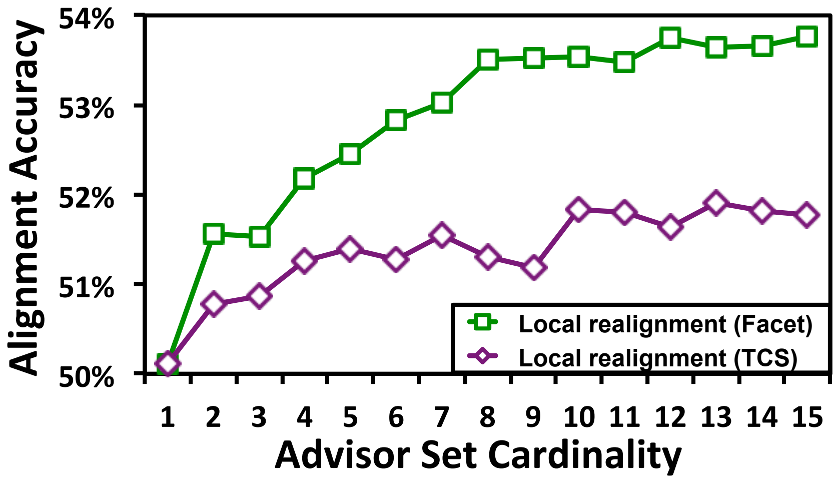


Figure 8.5: Accuracy of the default alignment and local realignment using `TCS` and `Facet` with various advisor set cardinalities. This figure compares the accuracy of alignments produced by the `Opal` default parameter settings applying local realignment using either the `TCS` or `Facet` estimator. The horizontal axis represents and increasing oracle advising set cardinality used for local realignment. The vertical axis shows the accuracy of the alignments produced by each of the advising methods averaged across difficulty bins.

8.3.4 Effect of iterating local realignment

The local advising process can be considered a refinement step for multiple sequence alignment. To continue refining the alignment we can iterate the local advising procedure (see Section 8.2.3). Iterating local advising should eventually reach some convergence in the alignment where you're no longer improving the result (i.e. the alignment regions are not being changed) or even worse you start deteriorating the result due to some noise in the accuracy estimator. To find the optimal iteration limit we ran all iteration limits from 1 to 25. We found that the peak accuracy on training benchmarks was at 5 iterations, and we use that for all other experiments shown in this chapter (other than those in Section 8.3.3). The table below shows the average accuracy of using local adaptive realignment on the default alignment with various number of iterations.

Iterations	1	2	3	4	5	10	15	25
Testing	53.5%	53.7%	54.1%	54.4%	54.5%	54.5%	54.5%	54.5%
Training	53.5%	53.9%	54.5%	54.6%	54.8%	54.8%	54.9%	54.9%

8.3.5 Summarizing the effect of adaptive local realignment

Table **??** summarizes how adaptive local realignment behaves across difficulty bins when used to modify alignments produced using the default parameter setting in `Opal`. The first two rows show how many of the 861 benchmarks are in each bin, as well as how many of them had at least one realignment region where the advisor chose to replace the global alignment. The fourth row shows the average number of *Bad* in a benchmark; on average about 2 regions were realigned for each default alignment. The last two rows summarize the percentage of the original columns those *Bad* regions covered, and how many of the columns from the original alignment ended up being replaced. In the easiest-to-align benchmark bin only 47% of the alignment columns were altered, while in the rest of the bins over 60% of the alignment columns were improved.

Table 8.1: Summary of Local Realignment on Default Alignments

Bin	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Overall**
Total number of benchmarks	12	12	20	34	26	50	61	74	137	434	861
Benchmarks unchanged	4	5	4	7	7	16	16	13	22	82	176
Benchmarks modified by adaptive local realignment	8	7	16	27	19	34	46	61	115	352	685
Percentage of benchmarks altered	67%	58%	80%	79%	73%	68%	74%	82%	84%	81%	80%
Average *Bad* regions per benchmark	1.92	2.17	2.50	1.88	2.23	2.14	2.31	2.16	2.48	2.19	2.23
Average percentage of original column realigned	75%	73%	76%	70%	75%	77%	74%	73%	75%	72%	73%
Average percentage of original column replaced	64%	60%	68%	60%	66%	72%	65%	63%	64%	47%	57%

8.3.6 Running time

As currently implemented in `Opal` local advising does not take advantage of the independence of the calls to the aligner in the parameter advising step and running them in parallel. Therefore we see a large increase in time consumption when generating locally advised alignments. In particular the average time for computing an alignment using the default global parameter setting goes from about 8 seconds to just over 36 seconds using an advisor set cardinality of $k = 10$. When iterating the local advising step 5 times we see the average running time increase to 110 seconds.

In contrast *global* advising exploits the independence of the aligner on different parameter settings. The running time for advisor set cardinality $k = 10$ for global advising alone is around 33 seconds, much less than the 10-fold increase to be expected if advising was not done in parallel. Even though global advising is done in parallel, local advising is not; the average running time over all benchmarks increases to 68 and 178 seconds for combining local and global advising, performing local advising on all global alignments with and without iteration, respectively.

8.3.7 Local and global advising in `Opal`

The development trunk of the `Opal` aligner includes the ability to perform adaptive local realignment both with and without parameter advising. To achieve the same results shown here the following commands were used to run the aligner:

(A) **Parameter advising with local realignment on all**

```
java opal.Opal --in <input file>\
   --facet_structure <structure file>\
   --configuration_file <parameter set>\
   --out_best <output file>
```

(B) **Parameter advising with local realignment on single**

```
java opal.Opal --in <input file>\
   --facet_structure <structure file>\
   --configuration_file <parameter set>\
   --out_prerealignment_best_realignment\
      <output file>
```

(C) **Parameter advising without local realignment**

```
java opal.Opal --in <input file>\
    --facet_structure <structure file>\
    --advising_configuration_file <set>\
    --out_best <output file>
```

(D) **Default alignment with local realignment**

```
java opal.Opal --in <input file>\
    --facet_structure <structure file>\
    --realignment_configuration_file <set>\
    --out <output file>
```

(E) **Default alignment without local realignment**

```
java opal.Opal --in <input file>\
    --out <output file>
```

Summary

We have presented *adaptive local realignment*, to our knowledge the first method that demonstrably boosts protein multiple sequence alignment accuracy by locally realigning regions that may have distinct mutation rates using different aligner parameter settings. Applying this new method alone to alignments initially computed using a single optimal default parameter setting already improves alignment accuracy significantly. When combined with methods to select an initial non-default parameter setting for the particular input sequences through global parameter advising, this new local parameter advising method greatly improves accuracy even further. We have also made available a tool that performs adaptive local realignment with the `Opal` aligner.

CHAPTER 9

Predicting Core Columns

Overview

In a computed multiple sequence alignment, the *coreness* of a column is the fraction of its substitutions that are in so-called core columns of the unknown reference alignment of the sequences, where the core columns of the reference alignment are those that are reliably correct. In the absence of knowing the reference alignment, the coreness of a column can only be estimated. We develop for the first time a column coreness estimator for protein multiple sequence alignments.

Our approach to predicting coreness is similar to nearest-neighbor classification from machine learning, except we transform nearest-neighbor distances into a coreness estimate using a regression function, and we automatically learn an appropriate distance function through a new optimization formulation that solves a large-scale linear programming problem. We apply our coreness estimator to improving *parameter advising*, the task of choosing good parameter values for an aligner's scoring function, and show that our estimator strongly outperforms others from the literature, providing a significant boost in advising accuracy.

9.1 Introduction

The accuracy of a multiple sequence alignment computed on a benchmark set of input sequences is usually measured with respect to a *reference alignment* that represents the gold-standard alignment of the sequences. For protein sequences, reference alignments are typically determined by structural superposition of the known three-dimensional structures of the proteins in the benchmark. The accuracy of a computed alignment is then defined to be the fraction of substitutions of pairs of residues in the so-called *core columns* of the reference alignment that are also present in columns of the computed alignment. Core columns represent those in the reference that are deemed to be reliable, and are columns containing a residue from every input sequence such that the pairwise distances between these residues in the structural superposition of the proteins are all within some threshold (typically a few angstroms). In short, given a known reference alignment whose columns are labeled as either core or non-core, we can determine the accuracy of any other computed alignment of its proteins by evaluating the fraction of substitutions in these core columns that are recovered. For a given column in a computed alignment, we can also define the *coreness* value of the column to be the fraction of its substitutions that are in core columns of the reference alignment. A coreness value of 1 means the column of the computed alignment corresponds to a core column of the reference alignment.

When aligning sequences in practice, obviously such a reference alignment is not known, and the accuracy of the computed alignment, or the coreness of its columns, must be estimated. A good *accuracy estimator* for computed alignments is extremely useful. It can be leveraged to pick among alternate alignments of the same sequences the one of highest estimated accuracy, for example, to choose good parameter values for an aligner's scoring function, called *parameter advising*; or to select the best result from a collection of different aligners, yielding a natural *ensemble aligner* that can be far more accurate than any individual aligner in the collection.

Similarly, a good *coreness estimator* for columns in a computed alignment can be used to mask out unreliable regions of the alignment before computing an evolutionary tree, or to improve an alignment accuracy estimator by concentrating its evaluation function on columns of higher estimated coreness, thereby boosting the performance of parameter advising. In fact, in principle a perfect coreness estimator would itself yield an ideal accuracy estimator.

In this chapter, we develop for the first time a column-coreness estimator for protein multiple sequence alignments. Our approach to predicting coreness is similar in some respects to nearest-neighbor classification from machine learning, except we transform nearest-neighbor distance into a coreness estimate using a regression function, and automatically learn an appropriate distance function through a new optimization formulation that solves a large-scale linear programming problem. We evaluate the performance of our new coreness estimator by applying it to the task parameter advising in multiple sequence alignment.

9.1.1 Related work

To our knowledge, this is the first fully general attempt to directly estimate the coreness of columns in computed protein alignments. In the literature, the `GUIDANCE` tool (Sela et al., 2015) gives reliability values for alignment columns, which they evaluate by measuring the classification accuracy of predicting totally correctly aligned core columns from reference alignments, though they do not attempt to relate reliability to coreness. `GUIDANCE` also requires alignments to contain at least four sequences, which limits the alignment benchmarks that can be considered. Tools are also available that assess the quality of columns in a multiple alignment, and can be categorized into those that compute a column quality score which can be thresholded, and those that only identify columns that are unreliable (for removal from further analysis). The popular quality score tools are `TCS` (Chang et al., 2014), `ZORRO` (Wu et al., 2012), and `Noisy` (Dress et al., 2008); these can be used to modify the feature functions in an accuracy estimator such as `Facet` (DeBlasio and Kececioglu, 2014b), as we later propose in Section 9.3.2. Tools that simply mask unreliable columns of an alignment include `ALISCORE` (Kück et al., 2010), `GBLOCKS` (Castresana, 2000), and `TrimAL` (Capella-Gutierrez et al., 2009).

We focus on comparing our coreness estimator to `TCS` and `ZORRO`, as these are the most recent tools that provide quality scores, as opposed to simply masking columns. Furthermore, of the above tools, `ALISCORE`, `GBLOCKS` and `GUIDANCE` have been shown to be dominated by `ZORRO`, while `Noisy` in turn has been shown to be dominated by `GUIDANCE`.

Plan of the chapter

In the next section, we present our method for learning a coreness estimator. Section 9.3 explains how we use predicted coreness to improve accuracy estimation for protein alignments. Section 9.4 then evaluates our approach to coreness prediction by applying the improved accuracy estimator to alignment parameter advising.

9.2 Learning a coreness estimator

To describe how we learn a column coreness estimator, we first discuss our *representation* of alignment columns, and our grouping of consecutive columns into *window classes*; we then present our *regression function* for estimating coreness, which transforms a distance to a window class into a coreness value; and finally, we describe how we learn this window distance function by solving a large-scale *linear programming* problem.

9.2.1 Representing alignment columns

We want to represent a multiple alignment column in a form that captures the association of amino acids and predicted secondary-structure types, but is independent of the number of sequences in the alignment. This is necessary for the labeled column examples in our training set to be useful for estimating the coreness of columns that come from other alignments with arbitrary numbers of sequences.

Let Σ be the 20-letter amino acid alphabet, and $\Gamma = \{\alpha, \beta, \gamma\}$ be the secondary-structure alphabet, corresponding respectively to types *α-helix*, *β-strand*, and *other* (also called *coil*). We encode the association of an amino acid $c \in \Sigma$ with its predicted secondary structure type $s \in \Gamma$ using an ordered pair (c, s) that we call a *state*, from the set $Q = (\Sigma \times \Gamma) \cup \{\xi\}$. Here $\xi = (\varepsilon, \varepsilon)$ is the *gap state*, where $\varepsilon \notin \Sigma$ is the alignment *gap character* (often displayed by the dash symbol '-').

We represent a multiple alignment column as a distribution over the set of states Q, which we call its *profile* (mirroring standard terminology (e.g., Durbin et al., 1998, p. 101)). We denote the profile C for a given column by a function $C(q)$

on states $q \in Q$ satisfying $C(q) \geq 0$ and $\sum_{q \in Q} C(q) = 1$. For a column $(c_1 c_2 \cdots c_k)$ in a multiple alignment of k sequences, with associated predicted secondary structure types $(s_1 \cdots s_k)$, where for a gap $c_i = \varepsilon$ the associated secondary structure type is also $s_i = \varepsilon$, its profile C is,

$$C(q) \quad := \quad \frac{1}{k} \left| \left\{ i \,:\, (c_i, s_i) = q \right\} \right| .$$

In other words, $C(q)$ is the relative frequency of state q in the column.

We generalize this to secondary structure predictions that for amino acid c_i give *confidences* $p_i(\alpha), p_i(\beta), p_i(\gamma)$ that the amino acid is in each of the three secondary structure types (where these confidences sum to 1), as follows. For state $q = (a, s) \neq \xi$, profile C is then,

$$C(q) \quad := \quad \frac{1}{k} \sum_{1 \leq i \leq k \,:\, c_i = a} p_i(s) .$$

In other words, $C(q)$ is now the normalized total confidence in state $q \neq \xi$. For gap state $q = \xi$, value $C(\xi)$ is the same as before.

9.2.2 Classes of column windows

The ground truth of whether a column in a reference alignment is core or non-core depends on whether the residues of the proteins in that column are sufficiently close in space in the structural superposition of the folded 3-dimensional structures of the proteins. This folded structure at a residue is not simply a function of the amino acid of the residue itself, or its secondary structure type, but is also a function of the nearby residues in the protein. Consequently, to estimate the coreness of a given column in a computed alignment, we need additional contextual information from nearby columns of the alignment.

We gather this additional context around a given column by forming a window of consecutive columns centered on the given column. Formally, a *column window* W of width $w \geq 1$ is a sequence of $2w + 1$ consecutive column profiles $W_{-w} \cdots W_{-1} W_0 W_{+1} \cdots W_{+w}$ centered around profile W_0.

We define the following set of *window classes* $\mathcal{C}$, depending on whether the columns in a labeled training window are known to be core or non-core with respect to their reference alignment. We denote a column labeled core by `C`, and a column labeled non-core by `N`. For window width $w = 1$, which we use in our experiments, such labeled windows can be described by strings of length 3 over alphabet $\{\mathtt{C}, \mathtt{N}\}$. The three classes of *core windows* are $\mathtt{CCC}, \mathtt{CCN}, \mathtt{NCC}$, and the three classes of *non-core windows* are $\mathtt{CNN}, \mathtt{NNC}, \mathtt{NNN}$. (A window is considered core or non-core depending on the label of its center column.) Together these six classes comprise set $\mathcal{C}$. We call the five classes with at least one core column `C` in the window as *structured classes*, and the one class with no core columns the single *unstructured class*, which we denote by the symbol $\perp = \mathtt{NNN}$.

Reference alignments explicitly label their columns as core or non-core. For computed alignments, which have a known reference alignment, we label a column as core or non-core depending on whether the true coreness value for the column is above a fixed threshold.

9.2.3 The coreness regression function

We learn an estimator for the coreness of a column by fitting a regression function that first measures the similarity between a window around the column and training examples of windows with known coreness, and then transforms this similarity into a coreness value.

We express the similarity between windows in terms of the similarity of their corresponding columns. To measure the similarity between columns we use a *distance function* d on pairs of column profiles A, B of the form,

$$d(A,B) \;\; := \sum_{p,q \,\in\, Q} A(p)\; B(q)\; \sigma(p,q)\,,$$

where $\sigma(p,q)$ is a substitution score that measures the dissimilarity between the pair of states p, q.

We extend this to a distance d on pairs of windows $V = V_{-w} \cdots V_w$ and $W = W_{-w} \cdots W_w$ by,

$$d(V,W) \;\; := \sum_{-w \,\le\, i \,\le\, +w} d_i(V_i, W_i)\,,$$

where the d_i are *positional* distance functions on column profiles. Function d_i is given by its *positional* substitution scores $\sigma_i(p, q)$. The positional σ_i can score dissimilarity higher at positions i at the center of the window, and lower toward the edge of the window.

Finally, we extend this to *class-specific* window distance functions d_c that are specific to each window class $c \in \mathcal{C} - \{\bot\}$. Function d_c is given by its class-specific positional profile distance functions $d_{c,i}$, which are in turn given by class-specific positional substitution scores $\sigma_{c,i}$.

The *regression function* that estimates the coreness of a column first forms a window W centered on the column, and then performs the following. To transform a distance to coreness we use two different functions: function f_{core} for core classes, and function f_{non} for non-core classes.

(1) (*Find distance to closest class*) Across all labeled training windows, in all structured window classes, find the training window that has smallest class-specific distance to W. Call this closest window V, its class c, and their distance $\delta = d_c(V, W)$.

(2) (*Transform distance to coreness*) If class c is a core class, return coreness value $f_{\text{core}}(\delta)$. Otherwise, return value $f_{\text{non}}(\delta)$.

We next explain how we efficiently find distance δ, and then describe the transform functions f.

Finding the distance to a class

To find the distance of a window W to a class c, we need to find the *nearest neighbor* of W among the set of training windows T_c in class c, namely $\operatorname{argmin}_{V \in T_c} \{ d_c(V, W) \}$. Finding the nearest neighbor through exhaustive search by explicitly evaluating $d_c(V, W)$ for every window V can be expensive when T_c is large (and cannot be avoided in the absence of exploitable properties of function d_c).

When the distance function is a *metric*, for which the key property is the *triangle inequality* (namely that $d(x, z) \le d(x, y) + d(y, z)$ for any three objects x, y, z), faster nearest neighbor search is possible. In this situation, in a preprocessing step we can first build a data structure over the set T_c, which then allows us to perform faster nearest neighbor searches on T_c for any query window W. One of the best data structures for nearest neighbor search under a metric is the *cover tree* of Beygelzimer et al. (2006). Theoretically, cover trees permit nearest neighbor searches over a set of n objects in $O(\log n)$ time, after constructing a cover tree in $O(n \log n)$ time, assuming that the intrinsic dimension of the set under metric d has a so-called bounded expansion constant (Beygelzimer et al., 2006). (For actual data, the expansion constant can be exponential in the intrinsic dimension.) In our experiments, for nearest neighbor search we use the recently-developed *dispersion tree* data structure of Woerner and Kececioglu (2016), which in extensive testing on scientific data is significantly faster in practice than cover trees.

We build a separate dispersion tree for each structured window class $c \in \mathcal{C} - \{\bot\}$ over its training set T_c using its distance function d_c, in a preprocessing step. To find the nearest neighbor to window W over all training windows $\mathcal{T} = \bigcup_c T_c$, we then perform a nearest neighbor search with W in each class dispersion tree, and merge these $|\mathcal{C}| - 1$ search results by picking the one with smallest distance to W.

Transforming distance to coreness

We use a sigmoid function to transform nearest neighbor distance into a coreness value. Once we have learned the distance functions d_c, as described in Section 9.2.4, we fit the transform function to empirical coreness values measured at the distances observed for example windows from our set of training windows, as follows. We sort the examples by their observed nearest neighbor distance, and at each observed distance δ, we collect the m adjacent examples whose distance is below δ, and the m adjacent examples above δ. We then compute the average true coreness value of these $2m+1$ examples, and assign this average true coreness value to distance δ. A sigmoid curve is then fit to these pairs of average true coreness and observed nearest neighbor distance values. This fitting process is performed separately for example windows from core classes, and non-core classes.

The particular sigmoid that we fit is the *logistic function.* The general form of the logistic function f that we use is,

$$f(x) \;:=\; \ell \;+\; (u-\ell)\,\frac{1}{1 \;+\; e^{ax+b}}\,,$$

with four parameters a, b, ℓ, u, where ℓ and u are respectively the minimum and maximum average true coreness values observed for the examples, and a and b are shape parameters. We use the curve fitting tools in `SciPy` (Jones et al., 2001) (which are a wrapper for `MINPACK` (Moré et al., 1984)) to find values for the shape parameters a, b that best fit the data.

We separately fit logistic functions $f_{\mathrm{core}}(\delta)$ and $f_{\texttt{non}}(\delta)$, with their own parameter values a, b, ℓ, u, to data from the core and non-core classes, respectively. For function f_{core}, shape parameter a is positive (so coreness is decreasing in the distance δ to a *core* class); for f_{non}, parameter a is negative (so coreness is increasing in the distance δ from a *non-core* class). As Figure 9.1 in Section 9.4.1 later shows, these logistic transform functions fit actual coreness data remarkably well.

9.2.4 Learning the distance function by linear programming

We now describe the linear program used to learn the distance function on column windows. The linear program learns a different, *class-specific*, distance function d_c for each window class $c \in \mathcal{C}$. These distance functions d_c are made commensurate between classes by a final rescaling step after solving the linear program.

Again we divide the window classes $\mathcal{C}$ into two categories: the *structured classes*, containing windows centered on core columns, or centered on non-core columns that are flanked on at least one side by core columns; and the *unstructured class*, containing windows of only non-core columns. We again denote this unstructured class of completely non-core windows by $\perp \in \mathcal{C}$.

In principle, the linear program tries to find distance functions d_c that would make the following "conceptual" nearest neighbor classifier accurate. (Note we are not actually learning such a classifier.) This conceptual classifier forms a window W centered on the column to be classified, and first finds the nearest neighbor to W over all structured classes $\mathcal{C}-\{\perp\}$ in the training set using their corresponding distance functions d_c. If the distance to this nearest neighbor is at most a threshold τ, the central column of window W is declared "core" or "non-core" depending on whether this nearest structured class c is core or non-core. Otherwise, the nearest neighbor distance exceeds threshold τ, the window is deemed to be in the unstructured non-core class $\perp$, and its central column is declared "non-core." The key aspect of this conceptual nearest neighbor classifier is that it can recognize a *completely non-core* window W from class $\perp$, without actually having any examples in its training set that are close to W. This is critical for our coreness estimation task, as the set of possible windows from the unstructured class $\perp$ is enormous and probably lacks any recognizable structure, which makes identifying them through having a near neighbor in the training set essentially hopeless. On the other hand, identifying windows from the structured classes is possible by having sufficiently many examples in the training set. The following linear program learns both distance functions d_c and such distance thresholds τ_c.

To construct the linear program, we partition the *training set* $\mathcal{T}$ of labeled windows by window class: subset $T_c \subseteq \mathcal{T}$ contains all training windows of class $c \in \mathcal{C}$. We then form a smaller *training sample* $S_c \subseteq T_c$ for each class c by choosing a random subset of T_c with a specified cardinality $|S_c|$.

The *constraints* of the linear program fall in several categories. For a sample training window $W \in S_c$, we identify other windows $V \in T_c$ from the same class c in the full training set that are close to W (under a default distance $\widetilde{d}_c$). We call these close windows V from the same class c, *targets*. Similarly for $W \in S_c$, we identify other windows $U \in T_b$ from a different class $b \neq c$ in the full training set that are also close to W (under $\widetilde{d}_b$). We call these other close windows U from a different class b, *impostors*. (This parallels the terminology of Weinberger and Saul (2009).) More formally, the *neighborhood* $\mathcal{N}_c(W,i)$ for a structured class $c \in \mathcal{C}-\{\perp\}$ denotes the set of i-nearest-neighbors to W (not including W) from training set T_c under the class-specific *default distance* function $\widetilde{d}_c$. (The default distance function that we use in our experiments is described in Section 9.4.1.) The constraints of the linear program find distance functions that for a sample window $W \in S_c$, *pull in* targets $V \in \mathcal{N}_c(W,i)$ by making $d_c(V,W)$ small, and *push away* impostors $U \in \mathcal{N}_b(W,i)$ for $b \neq c$ by making $d_b(U,W)$ large.

The neighborhoods $\mathcal{N}(W,i)$ that give the sets of targets and impostors for the linear programming formulation are defined with respect to default distance functions $\widetilde{d}$, as mentioned above. These neighborhoods really should be defined with respect to the learned distance functions d_c, but obviously they are not available until after the linear program is solved. We address this discrepancy by iteratively solving a series of linear programs. The first linear program at iteration 1 defines neighborhoods with respect to distance functions $d^{(0)} = \widetilde{d}$, and its solution yields the new functions $d^{(1)}$. In general, iteration i uses the previous iteration's functions $d^{(i-1)}$ to formulate a linear program whose solution yields the new distance functions $d^{(i)}$. This process is repeated for a fixed number of iterations, or until the change in the distance functions is sufficiently small.

The *target constraints* for each sample window $W \in S_c$ from each structured class $c \in \mathcal{C} - \{\perp\}$, and each target window $V \in \mathcal{N}_c(W,k)$, are,

$$e_{VW} \;\geq\; d_c(V,W) \;-\; \tau_c\,, \tag{9.1}$$

$$e_{VW} \;\geq\; 0\,, \tag{9.2}$$

where e_{VW} is a target *error variable* and τ_c is a *threshold variable.* In the above, quantity $d_c(V,W)$ is a linear expression in the *substitution score variables* $\sigma_{c,i}(p,q)$, so constraint (9.1) is a linear inequality in all these variables. Intuitively, we would like condition $d_c(V,W) \leq \tau_c$ to hold (so W will be considered to be in its correct class c); in the solution to the linear program, variable e_{VW} will equal $\max\big\{d_c(V,W) - \tau_c,\, 0\big\}$, the amount of error by which this ideal condition is violated.

In the target neighborhood $\mathcal{N}_c(W,k)$ above, parameter k specifies the number of targets for each sample window W. In our experiments we use a small number of targets, with $k = 2$ or 3.

The *impostor constraints* for each sample window $W \in S_c$ from each structured class $c \in \mathcal{C} - \{\perp\}$, and each impostor window $V \in \mathcal{N}_b(W, \ell)$ from each structured class $b \in \mathcal{C} - \{\perp\}$ with $b \neq c$, are,

$$f_W \;\; \geq \;\; \tau_b \; - \; d_b(V, W) \; + \; 1\,, \tag{9.3}$$

$$f_W \;\; \geq \;\; 0\,, \tag{9.4}$$

where f_W is an impostor error variable. Intuitively, we would like condition $d_b(V, W) > \tau_b$ to hold (so W will not be considered to be in the incorrect class b), which we can express by $d_b(V, W) \geq \tau_b + 1$ using a *margin* of 1. (Since the scale of the distance functions is arbitrary, we can always pick a unit margin without loss of generality.) In the solution to the linear program, variable f_W will equal $\max_{b \in \mathcal{C} - \{\perp\},\, V \in \mathcal{N}_b(W,\ell)} \big\{ \tau_b - d_b(V, W) + 1,\, 0 \big\}$, the largest amount of error by which this condition is violated for W across all b and V.

We also have impostor constraints for each completely non-core window $W \in T_\perp$, and each core window $V \in \mathcal{N}_b(W, \ell)$ from each structured core class b (as we do not want W to be considered core), which are of the same form as inequalities (9.3) and (9.4) above.

In the impostor neighborhood $\mathcal{N}_b(W, \ell)$ above, parameter ℓ specifies the number of impostors for each sample window W. We use a large number of impostors $\ell \approx 100$ in our experiments. Having a single impostor error variable f_W per sample window W (versus a target error variable e_{VW} for every W and target V) allows us to use a very large ℓ while still keeping the number of variables in the linear program tractable.

The *triangle inequality constraints*, for each structured class $c \in \mathcal{C} - \{\perp\}$, each window position $-w \leq i \leq w$, and all states $p, q, r \in Q$ (including the gap state ξ), are,

$$\sigma_{c,i}(p, r) \;\; \leq \;\; \sigma_{c,i}(p, q) \; + \; \sigma_{c,i}(q, r)\,. \tag{9.5}$$

These reduce to simpler inequalities when states p, q, r are not all distinct or coincide with the gap state (which we do not enumerate here to save space).

The remaining constraints, for all classes $c \in \mathcal{C}$, positions $-w \le i \le w$, states $p, q \in Q$, and gap state ξ, are,

$$\begin{aligned} \sigma_{c,i}(p,q) &= \sigma_{c,i}(q,p)\,, && (9.6)\\ \sigma_{c,i}(p,p) &\le \sigma_{c,i}(p,q)\,, && (9.7)\\ \sigma_{c,i}(p,q) &\ge 0\,, && (9.8)\\ \sigma_{c,i}(\xi,\xi) &= 0\,, && (9.9)\\ \tau_c &\ge 0\,, && (9.10) \end{aligned}$$

which ensure the distance functions are symmetric and non-negative. (We do not enforce the other metric conditions $d_c(W,W) = 0$ and $d_c(V,W) > 0$ for $V \ne W$, as these are not needed for our coreness estimation task, and we prefer having a less constrained distance d_c that might better minimize the following error objective.)

Finally, the *objective function* minimizes the average error over all training sample windows. Formally, we minimize,

$$\alpha \, \frac{1}{|\mathcal{C}|-1} \sum_{c \in \mathcal{C}-\{\perp\}} \frac{1}{|S_c|} \sum_{W \in S_c} \frac{1}{k} \sum_{V \in \mathcal{N}_c(W,k)} e_{VW} \quad +$$

$$(1-\alpha)\, \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \frac{1}{|S_c|} \sum_{W \in S_c} f_W\,,$$

where $0 \le \alpha \le 1$ is a blend parameter controlling the weight on target error versus impostor error. We note that in an optimal solution to this linear program, variables $e_{VW} = \max\{d_c(V,W) - \tau_c,\ 0\}$ and $f_W = \max_{V,b}\{\tau_b - d_b(V,W) + 1,\ 0\}$, since inequalities (9.1)–(9.4) ensure the error variables are at least these values, while minimizing the above objective function ensures they will not exceed them. Thus solving the linear program finds distance functions d_c, given by substitution scores $\sigma_{c,i}(p,q)$, that minimize the average over the training windows $W \in S_c$ of the amount of violation of our ideal conditions $d_c(V,W) \le \tau_c$ for targets $V \in T_c$ and $d_b(V,W) > \tau_b$ for impostors $V \in T_b$.

To summarize, the variables of the linear program are the substitution scores $\sigma_{c,i}(p,q)$, the error variables e_{VW} and f_W, and the threshold variables τ_c.

For n total training sample windows, k impostors per sample window, m window classes of width w, and amino acid alphabet size s, this is $\Theta(kn + s^2wm)$ total variables. The main constraints are the target constraints, impostor constraints, and triangle inequality constraints. For ℓ impostors per sample window, this is $\Theta\big((k+\ell m)n + s^3wm\big)$ total constraints. We ensure that solving the linear program is tractable by controlling the number ℓ of impostors and the total size n of the training sample.

After solving the linear program, we *rescale* the distance functions so their corresponding distance thresholds all match the common value $\tau := \max_{c \in \mathcal{C}} \tau_c$. Specifically, we scale up distance function d_c by multiplying its substitution scores $\sigma_{c,i}(p, q)$ (and distance threshold) by factor τ/τ_c. (Scaling up maintains that each class has a margin of at least 1.) This makes the distance functions d_c commensurate across classes. A conceptual 1-nearest-neighbor classifier for window W (which we do not employ) could then just find the nearest neighbor of W across all structured classes using their class-specific distance functions, say it is window V from class c, and classify W as a member of structured class c if $d_c(V, W) \le \tau$, and as a member of the unstructured non-core class $\perp$ otherwise. In actuality, rather than classifying W, we map its 1-nearest-neighbor distance $d_c(V, W)$ to a coreness value, as described in Section 9.2.3.

Ensuring the triangle inequality

We now show that the resulting distance functions satisfy the triangle inequality, which allows us to use fast data structures for metric-space nearest-neighbor search when evaluating the coreness estimator (as discussed in Section 9.2.3).

Theorem 8 (Triangle inequality on window distance) *The class distance functions d_c obtained by solving the linear program satisfy the triangle inequality.*

Proof For every class c, and all windows U, V, and W,

$$\begin{aligned}
d_c(U,W) &= \sum_i \sum_{p,r} U_i(p)\ W_i(r)\ \sigma_{c,i}(p,r) \\
&= \sum_i \sum_{p,q,r} U_i(p)\ V_i(q)\ W_i(r)\ \sigma_{c,i}(p,r) \qquad (9.11) \\
&\leq \sum_i \sum_{p,q,r} U_i(p)\ V_i(q)\ W_i(r) \cdot \Big(\sigma_{c,i}(p,q) \ + \ \sigma_{c,i}(q,r)\Big) \qquad (9.12) \\
&= \sum_i \sum_{p,q} U_i(p)\ V_i(q)\ \sigma_{c,i}(p,q) \ + \\
&\qquad\qquad \sum_i \sum_{q,r} V_i(q)\ W_i(r)\ \sigma_{c,i}(q,r) \qquad (9.13) \\
&= \sum_i d_{c,i}(U_i, V_i) \ + \ \sum_i d_{c,i}(V_i, W_i) \\
&= d_c(U,V) \ + \ d_c(V,W)\,,
\end{aligned}$$

where equation (9.11) follows from $\sum_q V_i(q) = 1$; inequality (9.12) follows from constraint (9.5) in the linear program; and equation (9.13) follows from $\sum_r W_i(r) = \sum_p U_i(p) = 1$. □

9.3 Using coreness to improve accuracy estimation

The `Facet` estimator (see Chapter 3) of alignment accuracy is a linear combination of efficiently-computable feature functions that are positively correlated with the true accuracy of an alignment. In general, the true accuracy of a computed alignment is evaluated just with respect to the columns of the reference alignment that are labeled as core; non-core columns do not contribute to true accuracy. Consequently, the ability to predict whether a column in a computed alignment corresponds to a core column in the unknown reference, or even better, to predict the coreness value of the column, should afford improved feature functions. We use the predicted coreness of computed alignment columns to improve the `Facet` estimator by: (1) creating a new feature function that attempts to directly estimate alignment accuracy by essentially counting the number of columns in the computed alignment that are predicted to be core and dividing by the estimated number of core columns in the reference, and (2) modifying the original feature functions so their evaluation is concentrated on columns with high predicted coreness. We first describe how we construct this new feature, and then briefly review the original features used in `Facet` and how we augment them with predicted coreness.

9.3.1 Creating a new coreness feature

The alignment accuracy measure known in the literature as "total column score" (or TC-score) is defined as the number of core columns in the reference alignment that are perfectly aligned in the computed alignment, divided by the number of core columns in the reference. Our new feature function, which we call *Predicted Alignment Coreness*, is designed to estimate the total column score of a computed alignment (which cannot be exactly determined as the correct reference alignment is unknown). Denote our *coreness estimator* from Section 9.2.3 for alignment column C by $\chi(C)$, which predicts coreness by evaluating our coreness regression function on a window centered on C. For a computed multiple sequence alignment $\mathcal{A}$ of a set of sequences $\mathcal{S}$, and a given coreness threshold κ, the Predicted Alignment Coreness feature function is,

$$F_{\texttt{AC}}(\mathcal{A}) \;:=\; \frac{\Big|\big\{C \in \mathcal{A} \;:\; \chi(C) \,\geq\, \kappa\big\}\Big|}{L(\mathcal{S})}\,, \tag{9.14}$$

where the numerator counts the number of columns of $\mathcal{A}$ whose predicted coreness is above threshold κ (in which case the column is effectively predicted as being core), and the normalization function L in the denominator is an estimate of the number of core columns in the unknown reference alignment of the sequences $\mathcal{S}$. The estimator L is a polynomial in several easily-computed quantities of sequences $\mathcal{S}$, whose coefficients are found by fitting L on benchmark sets of sequences for which a reference alignment (and the true number of core columns) is known.

We next describe how we determine estimator L.

Estimating the number of core columns

Function $L(\mathcal{S})$ that estimates the number of core columns in the reference alignment should tend to be increasing in the length of the sequences, and decreasing in their dissimilarity. The form of the estimator that we consider is a polynomial whose terms are generally the product of a measure of sequence length and a fractional quantity related to the percent identity of the sequences. We consider a variety of such measures, which gives a polynomial with many terms, and then solve a linear programming problem to find their coefficients by minimizing the L_1-norm to true core column counts on example benchmarks, which effectively selects the appropriate terms (since many coefficients turn out to be zero).

The length measures on sequence set $\mathcal{S}$ that we consider are the maximum, minimum, and average length of the sequences in set $\mathcal{S}$. We call $\mathcal{L}$ the set of these three length measures $\ell_{\max}$, $\ell_{\min}$, and ℓ_{avg}. The similarity measures on $\mathcal{S}$ that we consider are forms of percent identity, evaluated by summing over all pairs of sequences in $\mathcal{S}$ the maximum number of identities between each pair of sequences (computed by dynamic programming using the identity substitution matrix with no gap penalties), and normalizing by summing over all pairs of sequences the minimum, maximum, or average lengths of the sequences, giving percent identity measures $p_{\min}$, $p_{\max}$, and p_{avg}. We call $\mathcal{P}$ the set of these three percent identity measures. As a gap dissimilarity measure we also consider the difference in length between the longest and shortest sequences in $\mathcal{S}$ normalized by any of the length measures, giving the ratio measures $r_{\max}$, $r_{\min}$, and r_{avg}, as well as the length ratios $r_{\text{mm}} := \ell_{\min}/\ell_{\max}$, $r_{\text{am}} := \ell_{\text{avg}}/\ell_{\max}$, $r_{\text{ma}} := \ell_{\min}/\ell_{\text{avg}}$. Call $\mathcal{R}$ the set of these ratio measures.

The general form of estimator L is then,

$$\begin{aligned} L(S) \;:=\; & \sum_{\ell\in\mathcal{L},\, p\in\mathcal{P}} c_{\ell p}\; \ell(S)\; p(S) \;+\; \sum_{\ell\in\mathcal{L},\, r\in\mathcal{R}} c_{\ell r}\; \ell(S)\; r(S) \;+ \\ & \sum_{\ell\in\mathcal{L},\, p\in\mathcal{P},\, r\in\mathcal{R}} c_{\ell pr}\; \ell(S)\; p(S)\; r(S)\,. \end{aligned}$$

We fit coefficients $c_{\ell p}, c_{\ell r}, c_{\ell pr}$ by solving a linear program that minimizes the sum of the absolute values of the differences between the true number of core columns and the estimated number over all reference alignments in our suite of benchmarks.

The fitted function L that we use for evaluating the Predicted Alignment Coreness feature $F_{\texttt{AC}}$ is given in Section 9.4.1.

9.3.2 Augmenting former features by coreness

Since the true accuracy of a computed alignment is measured just with respect to the core columns of a reference alignment, and non-core columns are ignored, concentrating an accuracy estimator on columns with higher coreness should improve the estimator. Accordingly, we modify the alignment feature functions used by the `Facet` estimator (see Chapter 3) to focus their evaluation on columns of higher predicted coreness. Below we discuss only those features that can incorporate coreness; a full description of all feature functions in `Facet` is in Chapter 3.

Secondary Structure Blockiness takes secondary structure predictions from `PSIPRED` (Jones, 1999) and finds a packing of secondary structure blocks of maximum total score, where a block is an interval of columns and a subset of the sequences such that all residues in the block have the same secondary structure prediction, a packing is a set of blocks whose column intervals are disjoint, and the score of a block is the total number of pairs of residues within the columns in the block. We modify the score of a block by weighting the number of pairs per column by the column's predicted coreness. *Secondary Structure Identity* is the fraction of substitutions in the computed alignment that share the same predicted secondary structure, which we modify by weighting the count of substitutions with shared structure by their column's predicted coreness. *Amino Acid Identity* uses predicted coreness to weight the fraction of substitutions in a column that are in the same amino acid equivalence class. We modify *Average Substitution Score* by averaging the `BLOSUM62` score (Henikoff and Henikoff, 1992) of all substitutions, weighted by their column's predicted coreness.

9.4 Assessing the coreness prediction

We evaluate the performance of our new approach to core column prediction, and its use in accuracy estimation for alignment parameter advising, through experiments on a collection of protein multiple sequence alignment benchmarks. A full description of the benchmarks, and the universe U of parameter choices used for parameter advising, can be found in Chapter 3, and is briefly described here.

The benchmarks used in our experiments consist of reference alignments of protein sequences that are largely induced by structurally aligning their known three-dimensional structures. In particular, we use the `BENCH` suite of Edgar (2009), supplemented by a selection from the `PALI` suite of Balaji et al. (2001). The full benchmark collection we use consists of 861 reference alignments.

As is common in benchmark suites, easy-to-align benchmarks are highly over-represented in this collection. To correct for this bias towards easy benchmarks when evaluating average advising accuracy, we binned the 861 benchmarks by *hardness*, which we measured by the true accuracy of the alignment of the benchmark's sequences computed using the multiple alignment tool `Opal` under its optimal default parameter setting. We then divided the full range $[0, 1]$ of accuracies into 10 bins, where bin b for $b = 1, \ldots, 10$ contains hardness interval $\big((b-1)/10, b/10\big]$, and has 12, 12, 20, 34, 26, 50, 62, 74, 137, and 434 benchmarks, respectively.

We use *12-fold cross-validation* to assess the improvement in advising performance gained by learning the coreness estimator and our improved accuracy estimator. We construct training and testing subsets of the alignment benchmarks by evenly and randomly distributing benchmarks into 12 groups for each hardness bin; forming 12 splits of the entire collection of benchmarks into a training class and a testing class, where each split places one group in a bin into the training class and the other 11 groups in the bin into the training class; and for each split, generating a *training set* and *testing set* of example alignments by, for each benchmark B in a training or testing class, generating $|U|$ example alignments in the respective training or testing set by running `Opal` on B with each parameter choice in universe U.

An estimator learned on the examples from a training set was evaluated on examples from the corresponding testing set. The results we report are averages over 12 folds, where each *fold* is one of these pairs of associated training and testing sets. (Note that across the 12 folds, every example is tested on exactly once.)

9.4.1 Constructing the coreness regressor

We next present results on constructing the coreness regressor, specifically, on learning its distance function, mapping distances to coreness, and estimating the number of core columns.

Learning the distance function

The set of column windows for each class were constructed using the reference alignments of the benchmarks in the training set for each cross-validation fold. A subsampling of 4000 examples of each class was put into the set of training examples, and 4000 examples (or the remaining examples of that class, whichever is smaller) are put into the database for each class searched for nearest neighbors. We use a subset of 2000 of the 24,000 collected training examples for learning distances, to reduce the training time. A similar set of 2000 windows was collected from the alignments of testing benchmarks, to test the generalization of the distance functions when used for core column prediction.

We use a default distance between the training windows and each example window in the database for each class to get the initial sets of targets and impostors. The default distance on a pair of states is a linear combination of the `VTML200` amino acid substitution score (shifted and scaled to a dissimilarity value in the range $[0, 1]$) and the identity of the secondary structure prediction. For each column i with $-1 \leq i \leq 1$ in a window of three columns, we set the column weight w_i to $w_0 = \frac{1}{2}$, $w_{\{-1,+1\}} = \frac{1}{4}$ for all columns in a class c that are core, and $w_i = 0$ for non-core columns. The distance between states $p = (a, s)$ and $q = (b, t)$ in the ith column of class c is,

$$\sigma_{c,i}(p,q) \;=\; w_i \Big(\alpha \, \mathtt{VTML200}(a,b) \;+\; (1-\alpha)\,[s \neq t] \Big) \, ,$$

where $\alpha = \frac{1}{2}$, and expression $[s \neq t]$ evaluates to 1 if $s \neq t$ and 0 otherwise.

We then learn a distance function using these initial sets of targets and impostors. We use 2 targets and 150 impostors per training window per class. Once a distance function is learned, we can use it to recompute the sets of targets and impostors for learning a new distance function, and iterate this learning process. The table below shows the *area under the* receiver operating characteristic *curve* (AUC) for the first 10 iterations of distance learning, on both the training and testing examples. There is a steady increase in AUC on training examples for the first four iterations, with only a slight improvement in testing AUC; after the fifth iteration, no further improvement is seen.

Iteration	1	2	3	4	5	6	7	8	9	10
training AUC	86.3	93.9	98.9	99.3	99.3	99.4	99.3	99.3	99.3	99.4
testing AUC	83.8	82.5	84.9	84.8	85	84.8	84.6	84.6	84.6	84.3

We also performed this same training procedure using *random examples* from the correct and incorrect class databases for the targets and imposters. Using random targets and impostors, the training and testing AUC values were respectively 85.8 and 88.7 after a single iteration. While distance learning is effective, it is overfitting to the training data, most likely due to the small number of training examples used. Increasing the set of training examples led to prohibitively long running times for solving the linear program to find the optimal distance. Consequently, we use the distance functions learned on random points in our experiments that apply predicted coreness to improve the `Facet` estimator, as they generalize better.

Mapping distance to coreness

Figure 9.1 shows on its vertical axis the average true coreness of examples, superimposed with the fitted logistic transform function for predicted coreness, and on its horizontal axis the corresponding 1-nearest-neighbor distance, for one training fold of examples. The blue and red lines show the average coreness of the examples in the training set for which the nearest neighbor is in a core class and a structured non-core class, respectively. The top and bottom green curves show the two logistic transform functions for the core and non-core classes, respectively, fitted to this training data (which are used when predicting column coreness on testing data). Clearly the green logistic curves fit the data quite well. Note the steep transition from high to low coreness when the nearest neighbor is from a core class.

Estimating the number of core columns

For function $L(\mathcal{S})$ that estimates the number of core columns in the unknown reference alignment, the linear programming approach described in Section 9.3.1 to find optimal coefficients gave the fitted estimator,

$$\begin{aligned}
&(1.020)\, \ell_{\mathrm{min}}\, p_{\mathrm{max}}\, r_{\mathrm{mm}} \;+\; (0.151)\, \ell_{\mathrm{min}}\, r_{\mathrm{mm}} \;+ \\
&(0.035)\, \ell_{\mathrm{avg}}\, p_{\mathrm{max}}\, r_{\mathrm{am}} \;+\; (0.032)\, \ell_{\mathrm{avg}}\, p_{\mathrm{min}}\, r_{\mathrm{min}} \;+ \\
&(0.003)\, \ell_{\mathrm{max}}\, p_{\mathrm{avg}}\, r_{\mathrm{avg}}\,.
\end{aligned}$$

Figure 9.2 shows the correlation between the estimated number of core columns and the true number of core columns for each benchmark. The fitted estimator correlates well with the true number of core columns, but tends to overestimate, possibly due to larger benchmarks having columns that are very close to being core.

9.4.2 Improving parameter advising

The task of parameter advising is to select a choice of values for the parameters of the alignment scoring function for a multiple sequence alignment tool, based on the set of input sequences to align. A *parameter advisor* has two ingredients: (1) an *accuracy estimator*, which estimates the accuracy of a computed alignment (for which the reference is unavailable); and (2) an *advisor set*, which is the set of assignments of values to the aligner's parameters that are considered by the advisor. The advisor picks the choice of values from the advisor set for which the

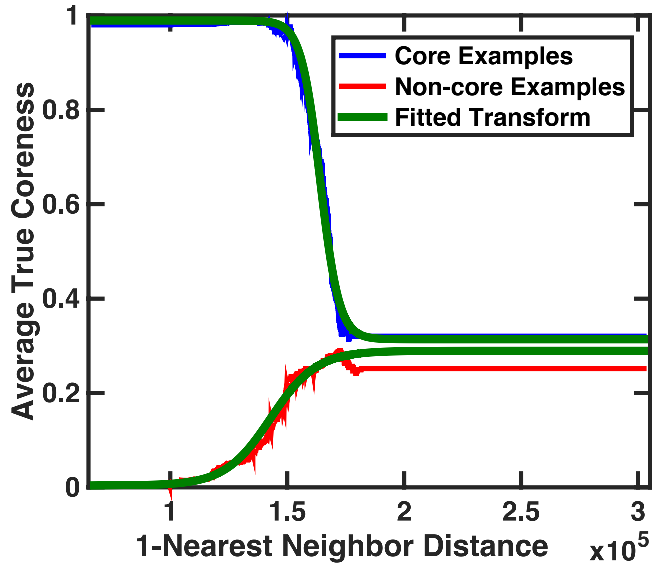


Figure 9.1: Fit of the logistic transform functions for the coreness regressor to the average true coreness of training examples at each nearest neighbor distance.

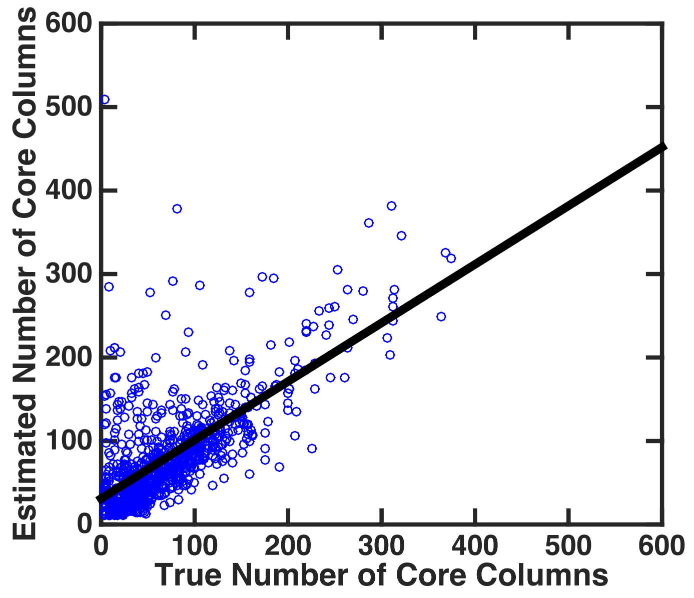


Figure 9.2: Correlation of the estimated number and true number of core columns.

aligner produces a computed alignment of the input sequences of highest estimated accuracy. In our experiments, we assess the performance of parameter advising using the `Facet` accuracy estimator modified by predicted coreness. For comparison, we also assess the advising accuracy of the `TCS` estimator, an unmodified version of `Facet`, and three versions of `Facet` modified using the column quality scores of `TCS`, `ZORRO`, and true coreness. We modify using true coreness to show a theoretical upper bound on the improvement possible if we could predict coreness perfectly.

We focus in this study on parameter advising for the multiple sequence alignment tool `Opal` (Wheeler and Kececioglu, 2007, 2012). While parameter advising increases the accuracy of many of the popular alignment tools (see Chapter 7), `Opal` is an ideal test bed for parameter advising, as in contrast to other aligners, it computes subalignments that are *optimal* with respect to the parameter choice for the sum-of-pairs scoring function at each node of the guide tree during progressive alignment.

The choice of advisor set is crucial for parameter advising. Clearly the performance of an advisor is limited by the quality of the parameter settings from which the advisor can pick. We consider two kinds of advisor sets (see Chapter 5): accuracy-estimator-independent *oracle sets*, which contain an optimal set of choices that maximize the performance of a perfect advisor that uses true accuracy for its accuracy estimator; and accuracy-estimator-dependent *greedy sets*, which tend to yield better performance in practice than oracle sets, but are tuned for a specific accuracy estimator. Finding such advisor sets requires specifying a finite universe of parameter choices from which to draw the set. Starting from roughly 16,900 parameter choices for `Opal`, we form a reduced universe by selecting the 25 most accurate parameter choices from each benchmark difficulty bin. This gave a universe of 243 parameter choices from which to construct oracle and greedy advisor sets.

When evaluating average advising accuracy on benchmarks, we correct for the over-representation of easy-to-align benchmarks by weighting benchmarks according to the same hardness bins described earlier. The weight of a benchmark falling in bin b is $(1/10)(1/n_b)$, where n_b is the number of benchmarks in bin b. These weights are such that each hardness *bin* contributes equally to the advising accuracy, which effectively uniformly averages advising accuracy across the full range of hardnesses.

Note that under this equal weighting of hardness bins, an advisor that uses only the single optimal default parameter choice will have an average advising accuracy of roughly 50% (illustrated later in Figure 9.3). This establishes as a point of reference an average advising accuracy of 50% as the *baseline* against which to compare advising performance.

Note that if we instead measured advising accuracy by uniformly averaging over *benchmarks*, then the predominance of easy benchmarks (for which little improvement is possible over the default parameter choice) makes both good and bad advisors tend to an average accuracy of nearly 100%. By uniformly averaging over *bins*, we can discriminate among advisors, though a typical value for average advising accuracy is now pulled down from 100% toward 50%.

Modifying the `Facet` accuracy estimator

We explore using our new coreness estimator, as well as `TCS` and `ZORRO`, to modify the existing features of `Facet` according to the procedure described in Section 9.3.2, and we also include the new Predicted Alignment Coreness feature described in Section 9.3.1. For the existing feature functions that can be modified by coreness, we consider using both the original and modified feature. We also explore using true coreness (as opposed to predicted coreness), which provides a theoretical limit on what is possible with a perfect coreness estimator. We learned coefficients for the feature functions of all these variants of `Facet` separately, using the difference-fitting technique described in Chapter 2.

The new alignment accuracy estimator that uses our coreness estimator has non-zero coefficients for seven features: our new feature, Predicted Alignment Coreness $F_{\texttt{AC}}$; two features that have been modified with predicted coreness, namely, Amino Acid Identity $F'_{\texttt{AI}}$ and Secondary Structure Identity $F'_{\texttt{SI}}$; and the four original features Gap Open Density $F_{\texttt{GO}}$, Secondary Structure Agreement $F_{\texttt{SA}}$, Amino Acid Identity $F_{\texttt{AI}}$, and Secondary Structure Blockiness $F_{\texttt{BL}}$. The fitted accuracy estimator that uses predicted coreness is,

$$\begin{aligned}&(0.512)\,F_{\texttt{GO}} \;+\; (0.304)\,F'_{\texttt{SI}} \;+\; (0.157)\,F_{\texttt{SA}} \;+\; (0.109)\,F_{\texttt{AI}} \;+\\ &(0.096)\,F_{\texttt{BL}} \;+\; (0.025)\,F'_{\texttt{AI}} \;+\; (0.013)\,F_{\texttt{AC}}\,.\end{aligned}$$

These feature functions have different ranges, so the magnitudes of the coefficients do not necessarily correspond to the importance of the features.

Improvement on oracle advisor sets

Figure 9.3 compares these various accuracy estimators in the context of parameter advising using estimator-independent *oracle* advisor sets (see Chapter 5). The horizontal axis is the cardinality of the advisor set, i.e. the number of parameter choices available to the advisor, while the vertical axis is average advising accuracy using various accuracy estimators. We compare the advising accuracy using different versions of `Facet`, as well as using the `TCS` accuracy estimator, on the same oracle sets, to isolate the effect each modification to the accuracy estimator has on advising performance. Using our new coreness predictor to modify the features of `Facet` increases the accuracy of parameter advising by as much as much as 3%, compared to the original unmodified version. This increase is in addition to the improvement of unmodified `Facet` over `TCS`, the next-best accuracy estimator in the literature.

Improvement on greedy advisor sets

The results from the preceding section show the effect of using different accuracy estimators on the same advisor sets of parameter choices. Here we show the effect of using different accuracy estimators on *greedy* advisor sets (see Chapter 5), which are near-optimal accuracy-estimator-dependent advisor sets that are designed to boost the advising accuracy when using a given accuracy estimator.

Figure 9.4 shows the advising accuracy using the `Facet` estimator modified by true coreness, predicted coreness, `TCS`, and `ZORRO`, using greedy advisor sets found specifically for each of these accuracy estimators. (Here each accuracy estimator is used with a different advisor set learned specifically for it by a greedy algorithm.)

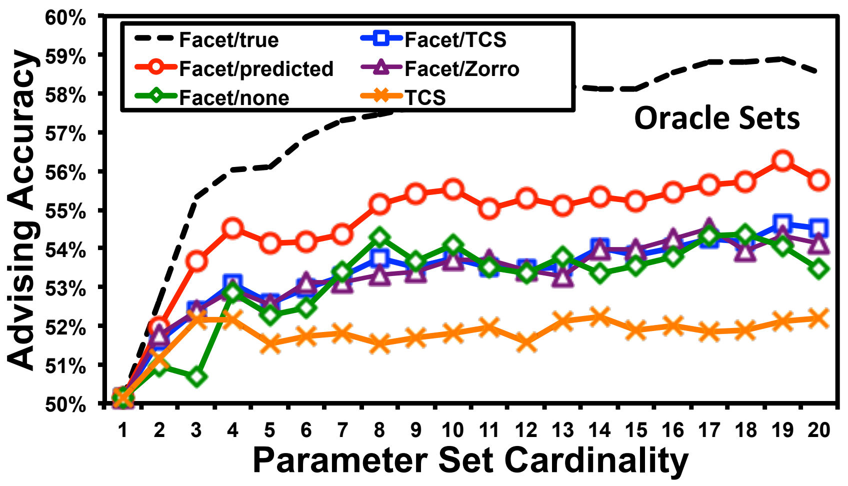


Figure 9.3: Advising accuracy using oracle sets with the modified `Facet` or `TCS` estimators.

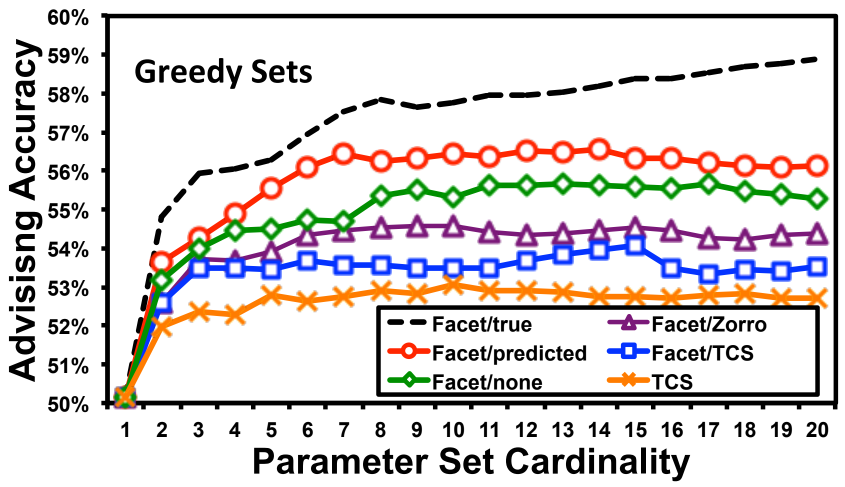


Figure 9.4: Advising accuracy using greedy sets with the modified `Facet` or `TCS` estimators.

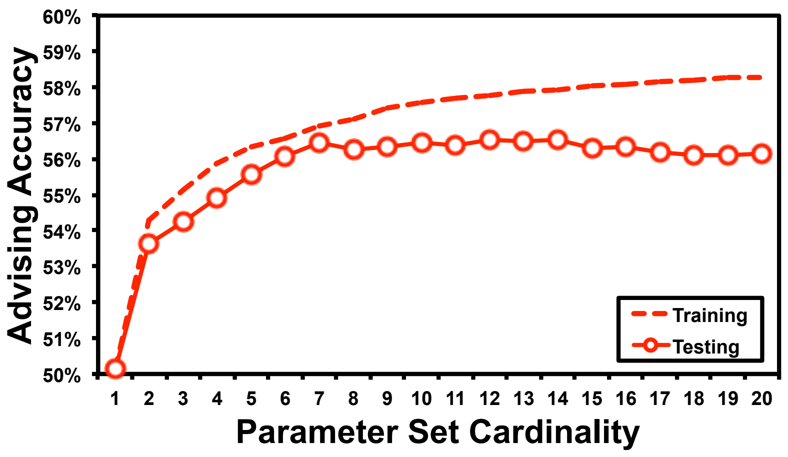


Figure 9.5: Training and testing advising accuracy using `Facet` with predicted coreness.

Once again the horizontal axis is the advisor set cardinality, and the vertical axis is advising accuracy averaged over the testing benchmarks in all folds. Using the new coreness predictor boosts the advising accuracy over using the original estimator by almost 2% when using a greedy advisor set of cardinality 7. In contrast, using `TCS` and `ZORRO` to modify features actually reduces the advising accuracy of greedy sets.

Figure 9.5 shows the advising accuracy on both training and testing benchmarks for the `Facet` estimator modified by predicted coreness using greedy advisor sets. The drop between training and testing accuracy suggests that by improving the generalization of greedy sets, further improvement in advising accuracy should be possible.

Summary

We have developed a column coreness estimator for protein multiple sequence alignments that uses a regression function on nearest neighbor distances for class distance functions learned by solving a new linear programming formulation. When applied to alignment accuracy estimation and parameter advising, the coreness estimator strongly outperforms others from the literature, and gives a significant boost in accuracy for advising.

CHAPTER 10

Conclusions

In this dissertation, we have addressed one of the major problems in protein multiple sequence alignment: how to choose a setting for the multitude of parameters in aligners. Multiple sequence alignment is an essential step in many biological analyses, and changing a parameter setting even slightly, can greatly affect the quality of the resulting alignment. Researchers typically use the default parameter settings that come with an aligner, which may be goof on average, but can nevertheless produce low quality alignments for particular inputs.

In Chapter 1, we introduced our approach called ***parameter advising***, which will find a parameter setting that yields a high quality alignment for a given input. A parameter advisor aligns the input sequences for each of a set of several parameter choices (where a parameter choice assigns a value to each of the aligner's tunable parameters), then selects the alignment of highest estimated accuracy from the resulting collection of alignments. A parameter advisor has two major components: (i) an *advisor set* of parameter choices considered by the advisor, and (ii) the *advisor estimator* that is used to rank the alignments produced by the aligner for these parameter choices.

In Chapters 2 and 3, we presented a new ***accuracy estimator*** called `Facet` (short for "feature-based accuracy estimator"), that computes an accuracy estimate as a linear combination of efficiently-computable feature functions. Multiple sequence alignment accuracy is measured as the fraction of aligned residue pairs in a known reference alignment that are recovered in the computed alignment. Without such a reference, alignment we are left to estimate accuracy. Chapter 2 described out framework for accuracy estimation and several techniques for finding coefficients that work well for parameter advising. Chapter 3 gave details on the feature functions used in `Facet`, which include novel features of an alignment that measure non-local properties.

Chapter 4 formally defined three problems: (1) **Optimal Advisor**, the problem of finding both the optimal coefficients for an estimator, and the optimal set of parameters choices for an advisor; (2) **Advisor Set**, a restriction of the Optimal Advisor problem where the objective is to find the set of parameter choices for a fixed estimator that has the highest average advising accuracy; and (3) **Advisor Estimator**, a restriction of the Optimal Advisor problem where the objective is to find the best estimator for a fixed advisor set. We show that all three problems are *NP-complete*.

Chapter 5 presented two approaches to the Advisor Set problem, in light of the fact that it is NP-complete. The first formulates the problem as an integer linear program. Unfortunately, finding its optimal solution is not practical even for small input sizes. The second is a greedy $\frac{\ell}{k}$***-approximation algorithm***, for any constant $\ell < k$. This greedy algorithm finds a near-optimal solution of cardinality k, given an optimal solution of size ℓ (which can be found in polynomial-time for constant ℓ).

We used the `Facet` accuracy estimator to perform parameter advising for the `Opal` (Wheeler and Kececioglu, 2007, 2012) multiple sequence aligner. Chapter 6 demonstrated parameter advising can greatly increases the accuracy of multiple sequence alignment for almost all inputs. This increase is most pronounced on the hardest benchmarks, where using an advising set of cardinality $k = 10$ boosts the accuracy by about 15%.

Chapter 7 presented the first true ***ensemble aligner***. Just as different parameter settings for a given input can produce very different alignments of the same sequences, the same is true for different aligners. These differences were exploited by extending parameter advising to *aligner advising*.

Since protein sequences do not necessarily have a homogeneous mutation rate across their length, the most accurate alignment for a set of input sequences may use different parameter settings in different regions of the alignment. Chapter 8 developed ***adaptive local realignment***, which identifies regions that may be misaligned under a single parameter choice and attempts to replace these regions with a more accurate alignment via parameter advising.

Chapter 9 presented a new approach to to predicting so-called ***core columns*** of a tertiary structure based benchmark within a computed alignment. Since core columns are the only locations where alignment accuracy is measured, ideally we would like to identify these locations when estimating accuracy so we can concentrate our estimator just on these positions. We use an approach similar to nearest-neighbor classification to construct a regression function that maps the amino-acid and predicted secondary-structure information in a window of columns into an estimate of how much of those windows come from core columns of the unknown reference alignment.

In addition to the developing the theory behind parameter advising, we have also produced software implementing of our `Facet` estimator for use by researchers. `Facet` is released as a stand-alone tool for parameter advising, as well as the necessary software to use a system for ensemble alignment. We have also released a new version of the `Opal` aligner that incorporates both parameter advising and adaptive local realignment. This enables any researcher who utilizes protein multiple sequence alignment to automatically increase the accuracy of their computed alignments without having to manually search for parameters that work well on their datasets.

10.1 Further research

This dissertation has developed a new methodology for parameter advising that can be applied outside multiple sequence alignment to any problem domain that has: (i) a tool, or set of tools, with multiple parameters whose values affect the accuracy of their resulting output, (ii) a collection of known ground truth results against which to measure the accuracy of the output of these tools, and (iii) some domain knowledge to discover feature functions that can be combined into an accuracy estimator.

One way to expand this work to new applications is to extend `Facet` to use on biological data beyond proteins. Extending the estimator to handle DNA and RNA sequences would allow advising to be used by a broader audience of researchers who employ multiple sequence alignment. Additionally this would allow parameter advising to be applied to new domains within bioinformatics such as sequence assembly and whole-genome alignment. Extending `Facet` to DNA and RNA alignments requires the creation of several feature functions that go beyond capturing only basic sequence information. As we have shown for proteins, the features that have the strongest correlation with true alignment accuracy are those that exploit the additional information gained from examining the predicted secondary-structure.

To create RNA feature functions we could also use predicted secondary structure, but this would only apply to the class of non-coding RNAs (ncRNA, those that form tertiary structure without coding for proteins). Just as with proteins, RNA secondary-structure can be predicted for a new input sequence. While this has been used to modify alignment objective functions (see DeBlasio et al., 2009, 2012a; DeBlasio, 2009), such approaches typically cannot handle pseudoknots (crossing secondary-structure pairings) when constructing alignments by dynamic programming. Since the features in `Facet` can take a global view of an RNA alignment, we could create feature functions that can account for pseudoknots.

To create DNA feature functions for `Facet`, there is no longer secondary-structure to exploit, but we might use some correlates that could guide us in predicting high accuracy alignments. One such additinal labeling (which is essentially what structure predictions provide) is to predict the categories of sequence regions. Such labels might include identifying protein-coding regions, translation start sites, potential ncRNAs, and so on. Another labeling that might help in estimating alignment accuracy could be predicting chromatin placement predictions. (When DNA is stored in chromosomes, it is wrapped around large proteins called chromatin, and only small regions between chromatin are accessible.) Recent work has shown that chromatin placement of these chromatin is used at certain times for translational regulation, so such locations might be conserved in high accuracy alignments. Beyond creating new feature functions for DNA, a major challenge is the lack of DNA multiple sequence alignment benchmarks. Without known ground-truth alignments, we cannot learn advisor sets and estimator coefficients. One recourse would be to generate simulated multiple sequence alignment benchmarks. With simulation, we know the true evolutionary history of a sequence, so we can recover the ground-truth alignment. Simulation, however, limits us to only learning simulated evolutionary parameter values and we may not learn the true biological parameter values that would yield the most biologically realistic alignments.

Once we have the `Facet` estimator for DNA, we can use it not only for global advising of DNA multiple sequence alignments, but also for local advising of whole-genome alignments through adaptive local realignment. Each section of a genome can evolve differently, and may even be rearranged or transposed from another sequence. Applying adaptive local realignment to whole-genome alignments could overcome the challenge posed by heterogeneity in genomic sequences.

De novo sequence assembly suffers from many of the same complications as multiple sequence alignment: a multitude of tools that can be used to align sequences, with each tool having many parameters that can affect the output of the assembler, and no good way to rank assemblies obtained by different methods. For sequence assembly, we again have to answer two questions: (1) What features can we create to measure the accuracy of an assembly? And (2) what is the ground-truth assembly? Standard measures of assembly quality are the N50 score, which measures the *length* of the smallest contig (a contiguous layout of sequence reads) that when placed in a sorted list of contigs covers half the expected length of the genome; and the L50 measure, which is the minimum *number* of contigs that cover half the genome. We could again devise new feature functions that measure the consistency of sequence labels (like those described earlier for DNA alignments). For the ground-truth assembly one possibility would be to employ read mapping against a known reference genome, and then remove the reference to leave a result resembling de novo assembly Alternatively, one could simulate a set of reads and take their known relation to the original underlying sequence as the ground-truth.

Another application of parameter advising within bioinformatics could be to read mapping: that is, mapping fragments to a known reference genome. The feature functions needed for read mapping might be different from those for de novo assembly, since we know the reference genome. A common quality measure is the fraction of reads successfully mapped, where reads can fail to be mapped due to sequencing error or a poor choice of the mapping parameters. Just as with DNA alignment and de novo assembly, there may be a problem with lack of benchmarks, but simulation might provide a ground-truth benchmarks.

In extending parameter advising to new applications, there are a few issues that still need to be be addressed. One is the generalization of our greedy estimator-aware advisor sets. Chapter 6 showed that greedy sets tend to not generalize well, and furthermore, exact sets generalize even worse. This behavior is exacerbated in the context of ensemble alignment in Chapter 7, where we compared default aligner advising to general aligner advising. Since the general advising universe is a *superset* of the default advising universe, as we increase the size of the universe, the advisor accuracy should increase (assuming we have a good estimator). In our experiments this is true for training data (see Figure 7.8), but it is not true when applied to testing data (see Figure 7.9).

One possible method to overcome the generalization issue for learning advisor sets could be to utilize inverse parametric alignment from Kim and Kececioglu (2008). Inverse parametric alignment find the optimal single choice of aligner parameters that give the highest average alignment accuracy for a set of examples. In the methods presented in this dissertation advisor set finding relies on a fixed universe of advisor parameter choices. This universe is intended to cover the entire range of possible values for all of the tunable parameters for an aligner. Many of the tunable parameters are continuous so we are forced to discretize the range of possible settings to generate a finite universe. As the granularity of this discretization is increased so too are the changes for overfitting assuming we keep the same set of benchmark alignments. Rather than relying on a fixed universe we can use inverse parametric alignment in a greedy manner to develop an advisor set. Start by using inverse parametric alignment to generate single parameter choice, P_1, that is optimal on average for all of the available examples. Then remove any example that when aligned using P_1 already have high accuracy, essentially removing the high accuracy bin's examples in earlier chapters. With the remaining examples, which we know have low accuracy when aligned using P_1, we use inverse parametric alignment again to find the optimal parameters, P_2. We continue this procedure until either we have a certain number of parameters or there are no longer any low accuracy examples. We then construct an advisor set as the union of the parameter choices we encountered

$$\mathcal{P} = P_1 \, \cup \, P_2 \, \cup \, \cdots \, \cup P_k.$$

Using this method the learned advisor's accuracy is no longer limited by the predefined parameter universe.

We also have a similar issue with generalization in core-column prediction. While considerable work was done in Chapter 9 to lean good distance functions on training data for the nearest-neighbor search in out approach to coreness prediction, the distance function learned in that chapter were limited by the available computational resources for project. With more computing time, we may be able to learn better distance functions that yield a more accurate coreness predictor with the methods already developed. Additionally, we might also apply other distance metric learning techniques from machine learning, including kernel transformations on the input data to reduce its dimensionality.

Clearly there are many prosing directions in which to take this new methodology of parameter advising.

REFERENCES

Ahola, V., T. Aittokallio, M. Vihinen, and E. Uusipaikka (2006). A statistical score for assessing the quality of multiple sequence alignments. ***BMC Bioinformatics***, **7**(484), pp. 1–19.

Ahola, V., T. Aittokallio, M. Vihinen, and E. Uusipaikka (2008). Model-based prediction of sequence alignment quality. ***Bioinformatics***, **24**(19), pp. 2165–2171.

Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman (1990). Basic local alignment search tool. ***Journal of Molecular Biology***, **215**(3), pp. 403–410.

Aniba, M. R., O. Poch, A. Marchler-Bauer, and J. D. Thompson (2010). `AlexSys`: a knowledge-based expert system for multiple sequence alignment construction and analysis. ***Nucleic Acids Research***, **38**(19), pp. 6338–6349.

Anson, E. L. and E. W. Myers (1997). ReAligner: a program for refining DNA sequence multi-alignments. ***Journal of Computational Biology***, **4**(3), pp. 369–83.

Apweiler, R., A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L. S. L. Yeh (2004). `UniProt`: the Universal Protein knowledgebase. ***Nucleic Acids Research***, **32**(Database), pp. D115–D119.

Armougom, F., S. Moretti, V. Keduas, and C. Notredame (2006). The `iRMSD`: a local measure of sequence alignment accuracy using structural information. In ***Bioinformatics***, pp. E35–E39.

Bahr, A., J. D. Thompson, J. C. Thierry, and O. Poch (2001). `BAliBASE` (Benchmark Alignment dataBASE): enhancements for repeats, transmembrane sequences and circular permutations. ***Nucleic Acids Research***, **29**(1), pp. 323–326.

Balaji, S., S. Sujatha, S. S. C. Kumar, and N. Srinivasan (2001). `PALI`: a database of Phylogeny and ALIgnment of homologous protein structures. ***Nucleic Acids Research***, **29**(1), pp. 61–65.

Berman, H. M., J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne (2000). The Protein Data Bank. ***Nucleic Acids Research***, **28**(1), pp. 35–242.

Beygelzimer, A., S. Kakade, and J. Langford (2006). Cover trees for nearest neighbor. ***Proceedings of the 23rd International Conference on Machine Learning (ICML)***, pp. 97–104.

Bradley, R. K., A. Roberts, M. Smoot, S. Juvekar, J. Do, C. Dewey, I. Holmes, and L. Pachter (2009). Fast Statistical Alignment. ***PLoS Computational Biology***, **5**(5), pp. 1–15.

Bucka-Lassen, K., O. Caprani, and J. Hein (1999). Combining many multiple alignments in one improved alignment. ***Bioinformatics***, **15**(2), pp. 122–130.

Camon, E., M. Magrane, D. Barrell, V. Lee, E. Dimmer, J. Maslen, D. Binns, N. Harte, R. Lopez, and R. Apweiler (2004). The Gene Ontology Annotation (GOA) Database: sharing knowledge in Uniprot with Gene Ontology. ***Nucleic Acids Research***, **32**(90001), pp. 262D–266.

Capella-Gutierrez, S., J. M. Silla-Martinez, and T. Gabaldón (2009). `trimAl`: a tool for automated alignment trimming in large-scale phylogenetic analyses. ***Bioinformatics***, **25**(15), pp. 1972–1973.

Carrillo, H. and D. Lipman (1988). The Multiple Sequence Alignment Problem in Biology. ***SIAM Journal on Applied Mathematics***, **48**(5), pp. 1073–1082.

Castresana, J. (2000). Selection of conserved blocks from multiple alignments for their use in phylogenetic analysis. ***Molecular Biology and Evolution***, **17**(4), pp. 540–552.

Chang, J. M., P. D. Tommaso, and C. Notredame (2014). `TCS`: a new multiple sequence alignment reliability measure to estimate alignment accuracy and improve phylogenetic tree reconstruction. ***Molecular Biology and Evolution***, **31**(6), pp. 1625–1637.

Collingridge, P. W. and S. Kelly (2012). `MergeAlign`: improving multiple sequence alignment performance by dynamic reconstruction of consensus multiple sequence alignments. ***BMC Bioinformatics***, **13**(117), pp. 1—10.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). ***Introduction to Algorithms, Third Edition***. The MIT Press, 3rd edition. ISBN 0262033844, 9780262033848.

Darling, A. C., B. Mau, F. R. Blattner, and N. T. Perna (2004). `Mauve`: Multiple Alignment of Conserved Genomic Sequence With Rearrangements. ***Genome Research***, **14**(7), pp. 1394–1403.

Dayhoff, M. O., R. M. Schwartz, and B. C. Orcutt (1978). A model of evolutionary change in proteins. ***In Atlas of Protein Sequences and Structure***, **5**, pp. 345–352.

DeBlasio, D., J. Bruand, and S. Zhang (2009). `PMFastR`: A New Approach to Multiple RNA Structure Alignment. ***Proceedings of the 9th International Conference on Algorithms in Bioinformatics (WABI'09)***, pp. 49–61.

DeBlasio, D., J. Bruand, and S. Zhang (2012a). A Memory Efficient Method for Structure-Based RNA Multiple Alignment. ***IEEE/ACM Transactions on Computational Biology and Bioinformatics***, **9**(1), pp. 1–11.

DeBlasio, D. and J. Kececioglu (2014a). Learning Parameter Sets for Alignment Advising. ***Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB)***, pp. 230–239.

DeBlasio, D. and J. Kececioglu (2015). Ensemble Multiple Sequence Alignment via Advising. ***Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB)***, pp. 452–461.

DeBlasio, D. F. (2009). ***New Computational Approaches For Multiple RNA Alignment And RNA Search***. Masters Thesis. University of Central Florida, Orlando, Florida.

DeBlasio, D. F. and J. D. Kececioglu (2014b). `Facet`: software for accuracy estimation of protein multiple sequence alignments (version 1.1). `http://facet.cs.arizona.edu`.

DeBlasio, D. F. and J. D. Kececioglu (2016). Learning Parameter-Advising Sets for Multiple Sequence Alignment. ***IEEE/ACM Transactions on Computational Biology and Bioinformatics***. To appear.

DeBlasio, D. F., T. J. Wheeler, and J. D. Kececioglu (2012b). Estimating the accuracy of multiple alignments and its use in parameter advising. ***Proceedings of the 16th Conference on Research in Computational Molecular Biology (RECOMB)***, pp. 45–59.

Do, C. B., M. S. P. Mahabhashyam, M. Brudno, and S. Batzoglou (2005). `ProbCons`: probabilistic consistency-based multiple sequence alignment. ***Genome Research***, **15**(2), pp. 330–340.

Dress, A. W., C. Flamm, G. Fritzsch, S. Grünewald, M. Kruspe, S. J. Prohaska, and P. F. Stadler (2008). `Noisy`: Identification of problematic columns in multiple sequence alignments. ***Algorithms for Molecular Biology***, **3**(7), pp. 1–10.

Durbin, R., S. R. Eddy, A. Krogh, and G. Mitchison (1998). ***Biological Sequence Analysis: Probablistic Models of Proteins and Nucleic Acids***. Cambridge University Press.

Edgar, R. C. (2004a). `MUSCLE`: multiple sequence alignment with high accuracy and high throughput. ***Nucleic Acids Research***, **32**(5), pp. 1792–1797.

Edgar, R. C. (2004b). `MUSCLE`: a multiple sequence alignment method with reduced time and space complexity. ***BMC Bioinformatics***, **5**(113), pp. 1—19.

Edgar, R. C. (2009). `BENCH`. `http://www.drive5.com/bench`.

Estabrook, G., C. Johnson, and F. M. Morris (1975). An idealized concept of the true cladistic character. ***Mathematical Biosciences***, **23**(3), pp. 263 – 272.

Feng, D.-F. and R. F. Doolittle (1987). Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. ***Journal of Molecular Evolution***, **25**(4), pp. 351–360.

Finn, R. D., J. Mistry, J. Tate, P. Coggill, A. Heger, J. E. Pollington, O. L. Gavin, P. Gunasekaran, G. Ceric, K. Forslund, L. Holm, E. L. L. Sonnhammer, S. R. Eddy, and A. Bateman (2009). The Pfam protein families database. ***Nucleic Acids Research***, **38**(Database), pp. D211–D222.

Fitch, W. M. and E. Margoliash (1967). A method for estimating the number of invariant amino acid coding positions in a gene using cytochrome c as a model case. ***Biochemical Genetics***, **1**(1), pp. 65–71.

Garey, M. R. and D. S. Johnson (1979). ***Computers and Intractability: A Guide to the Theory of NP-completeness***. W.H. Freeman and Company, New York.

Gotoh, O. (1982). An improved algorithm for matching biological sequences. ***Journal of Molecular Biology***, **162**(3), pp. 705–508.

Gotoh, O. (1993). Optimal alignment between groups of sequences and its application to multiple sequence alignment. ***Computer Applications in the Biosciences***, **9**(3), pp. 361–370.

Gusfield, D. (1997). ***Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology***. Cambridge University Press, New York, NY, USA.

Henikoff, S. and J. G. Henikoff (1992). Amino acid substitution matrices from protein blocks. ***Proceedings of the National Academy of Sciences USA***, **89**(22), pp. 10915–10919.

Hertz, G. Z. and G. D. Stormo (1999). Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. ***Bioinformatics***, **15**(7-8), pp. 563–577.

Jones, D. T. (1999). Protein secondary structure prediction based on position-specific scoring matrices. ***Journal of Molecular Biology***, **292**(2), pp. 195–202.

Jones, E., T. Oliphant, P. Peterson, et al. (2001). `SciPy`: Open source scientific tools for Python. `http://www.scipy.org`.

Karlin, S. and S. F. Altschul (1990). Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. ***Proceedings of the National Academy of Sciences of the United States of America***, **87**(6), pp. 2264–2268.

Katoh, K., K.-i. Kuma, H. Toh, and T. Miyata (2005). `MAFFT` version 5: improvement in accuracy of multiple sequence alignment. ***Nucleic Acids Research***, **33**(2), pp. 511–518.

Katoh, K., K. Misawa, K.-i. Kuma, and T. Miyata (2002). `MAFFT`: a novel method for rapid multiple sequence alignment based on fast Fourier transform. ***Nucleic Acids Research***, **30**(14), pp. 3059–3066.

Kececioglu, J. and D. DeBlasio (2013). Accuracy estimation and parameter advising for protein multiple sequence alignment. ***Journal of Computational Biology***, **20**(4), pp. 259–279.

Kececioglu, J. and E. Kim (2006). Simple and Fast Inverse Alignment. ***Proceedings of the 10th Conference on Research in Computational Molecular Biology (RECOMB)***, pp. 441–455.

Kececioglu, J. and D. Starrett (2004). Aligning alignments exactly. In ***Proceedings of the 8th Conference on Research in Computational Molecular Biology (RECOMB)***, pp. 85–96. ACM.

Kemena, C., J.-F. Taly, J. Kleinjung, and C. Notredame (2011). `STRIKE`: evaluation of protein MSAs using a single 3D structure. ***Bioinformatics***, **27**(24), pp. 3385–3391.

Kim, E. and J. Kececioglu (2008). Learning Scoring Schemes for Sequence Alignment from Partial Examples. ***IEEE/ACM Transactions on Computational Biology and Bioinformatics***, **5**(4), pp. 546–556.

Kim, J. and J. Ma (2011). `PSAR`: measuring multiple sequence alignment reliability by probabilistic sampling. ***Nucleic Acids Research***, **39**(15), pp. 6359–6368.

Kück, P., K. Meusemann, J. Dambach, B. Thormann, B. M. von Reumont, J. W. Wägele, and B. Misof (2010). Parametric and non-parametric masking of randomness in sequence alignments can be improved and leads to better resolved trees. ***Frontiers in Zoology***, **7**(10), pp. 1–12.

Kuznetsov, I. B. (2011). Protein sequence alignment with family-specific amino acid similarity matrices. ***BMC Research Notes***, **4**(296), pp. 1–10.

Landan, G. and D. Graur (2007). Heads or tails: a simple reliability check for multiple sequence alignments. ***Molecular Biology and Evolution***, **24**(6), pp. 1380–1383.

Larkin, M. A. et al. (2007). `ClustalW` and `ClustalX` version 2.0. ***Bioinformatics***, **23**(21), pp. 2947–2948.

Lassmann, T. and E. Sonnhammer (2005a). `Kalign`: an accurate and fast multiple sequence alignment algorithm. ***BMC Bioinformatics***, **6**(298), pp. 1–9.

Lassmann, T. and E. L. L. Sonnhammer (2005b). Automatic assessment of alignment quality. ***Nucleic Acids Research***, **33**(22), pp. 7120–7128.

Lee, C., C. Grasso, and M. F. Sharlow (2002). Multiple sequence alignment using partial order graphs. ***Bioinformatics***, **18**(3), pp. 452–464.

Liu, K., T. J. Warnow, M. T. Holder, S. M. Nelesen, J. Yu, A. P. Stamatakis, and C. R. Linder (2011). `SATé-II`: Very Fast and Accurate Simultaneous Estimation of Multiple Sequence Alignments and Phylogenetic Trees. ***Systematic Biology***, **61**(1), pp. 90–106.

Liu, Y., B. Schmidt, and D. L. Maskell (2010). `MSAProbs`: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities. ***Bioinformatics***, **26**(16), pp. 1958–1964.

Loytynoja, A. and N. Goldman (2005). An algorithm for progressive multiple alignment of sequences with insertions. ***Proceedings of the National Academy of Sciences***, **102**(30), pp. 10557–10562.

Misof, B. and K. Misof (2009). A Monte Carlo approach successfully identifies randomness in multiple sequence alignments: a more objective means of data exclusion. ***Systematic biology***, **58**(1), pp. 21–34.

Moré, J. J., D. C. Sorensen, K. E. Hillstrom, and B. S. Garbow (1984). The `MINPACK` Project. ***Sources and Development of Mathematical Software***, pp. 88–111.

Muller, J., C. J. Creevey, J. D. Thompson, D. Arendt, and P. Bork (2010). `AQUA`: automated quality improvement for multiple sequence alignments. ***Bioinformatics***, **26**(2), pp. 263–265.

Müller, T., R. Spang, and M. Vingron (2002). Estimating amino acid substitution models: a comparison of Dayhoff's estimator, the resolvent approach and a maximum likelihood method. ***Molecular Biology and Evolution***, **19**(1), pp. 8–13.

Needleman, S. B. and C. D. Wunsch (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. ***Journal of Molecular Biology***, **48**(3), pp. 443–453.

Notredame, C., D. G. Higgins, and J. Heringa (2000). `T-Coffee`: A novel method for fast and accurate multiple sequence alignment. ***Journal of Molecular Biology***, **302**(1), pp. 205–217.

Notredame, C., L. Holm, and D. G. Higgins (1998). `COFFEE`: an objective function for multiple sequence alignments. ***Bioinformatics***, **14**(5), pp. 407–422.

Ortuño, F., O. Valenzuela, H. e. Pomares, and I. Rojas (2013). Evaluating Multiple Sequence Alignments Using a LS-SVM Approach with a Heterogeneous Set of Biological Features. ***Proceedings of the 12th International Work-Conference on Artificial Neural Networks (IWANN 2013)***, pp. 150–158.

Ortuno, F. M., O. Valenzuela, H. Pomares, F. Rojas, J. P. Florido, J. M. Urquiza, and I. Rojas (2012). Predicting the accuracy of multiple sequence alignment algorithms by using computational intelligent techniques. ***Nucleic Acids Research***, **41**(1), pp. e26–e26.

Pei, J. and N. V. Grishin (2001). `AL2CO`: calculation of positional conservation in a protein sequence alignment. ***Bioinformatics***, **17**(8), pp. 700–712.

Pei, J. and N. V. Grishin (2006). `MUMMALS`: multiple sequence alignment improved by using hidden Markov models with local structural information. ***Nucleic Acids Research***, **34**(16), pp. 4364–4374.

Pei, J. and N. V. Grishin (2007). `PROMALS`: towards accurate multiple sequence alignments of distantly related proteins. ***Bioinformatics***, **23**(7), pp. 802–808.

Pei, J., R. Sadreyev, and N. V. Grishin (2003). `PCMA`: fast and accurate multiple sequence alignment based on profile consistency. ***Bioinformatics***, **19**(3), pp. 427–428.

Penn, O., E. Privman, G. Landan, D. Graur, and T. Pupko (2010). An alignment confidence score capturing robustness to guide tree uncertainty. ***Molecular Biology and Evolution***, **27**(8), pp. 1759–1767.

Prakash, A. and M. Tompa (2009). Assessing the Discordance of Multiple Sequence Alignments. ***IEEE/ACM Transactions on Computational Biology and Bioinformatics***, **6**(4), pp. 542–551.

Raghava, G., S. M. Searle, P. C. Audley, J. D. Barber, and G. J. Barton (2003). `OXBench`: A benchmark for evaluation of protein multiple sequence alignment accuracy. ***BMC Bioinformatics***, **4**(1), pp. 1–23.

Ren, J. (2014). SVM-based Automatic Annotation of Multiple Sequence Alignments. ***Journal of Computers***, **9**(5), pp. 1109–1116.

Roshan, U. and D. R. Livesay (2006). `PROBALIGN`: multiple sequence alignment using partition function posterior probabilities. ***Bioinformatics***, **22**(22), pp. 2715–2721.

Roskin, K. M., B. Paten, and D. Haussler (2011). Meta-Alignment with `Crumble` and `Prune`: Partitioning very large alignment problems for performance and parallelization. ***BMC Bioinformatics***, **12**(1), pp. 1–12.

S. Schwartz, A. and L. Pachter (2007). Multiple alignment by sequence annealing. ***Bioinformatics***, **23**(2), pp. e24–e29.

Sela, I., H. Ashkenazy, K. Katoh, and T. Pupko (2015). `GUIDANCE2`: accurate detection of unreliable alignment regions accounting for the uncertainty of multiple parameters. ***Nucleic Acids Research***, **43**(W1), pp. W7–W14.

Sievers, F. et al. (2011). Fast, scalable generation of high-quality protein multiple sequence alignments using `Clustal Omega`. ***Molecular Systems Biology***, **7**(1), pp. 539–539.

Subramanian, A. R., M. Kaufmann, and B. Morgenstern (2008). `DIALIGN-TX`: greedy and progressive approaches for segment-based multiple sequence alignment. ***Algorithms for Mol. Biology***, **3**(6), pp. 1–11.

Subramanian, A. R., J. Weyer-Menkhoff, M. Kaufmann, and B. Morgenstern (2005). `DIALIGN-T`: An improved algorithm for segment-based multiple sequence alignment. ***BMC Bioinformatics***, **6**(66), pp. 1–13.

Suzek, B. E., H. Huang, P. McGarvey, R. Mazumder, and C. H. Wu (2007). `UniRef`: comprehensive and non-redundant UniProt reference clusters. ***Bioinformatics***, **23**(10), pp. 1282–1288.

IBM Corporation (2015). CPLEX: High-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming (version 12.6.2.0). http://www.ilog.com/products/cplex.

The UniProt Consortium (2007). The Universal Protein Resource (UniProt). ***Nucleic Acids Research***, **35**(suppl 1), pp. D193–D197.

Thompson, J. D., D. G. Higgins, and T. J. Gibson (1994). ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. ***Nucleic Acids Research***, **22**(22), pp. 4673–4680.

Thompson, J. D., F. Plewniak, R. Ripp, J.-C. Thierry, and O. Poch (2001). Towards a reliable objective function for multiple sequence alignments. ***Journal of Molecular Biology***, **314**(4), pp. 937–951.

Thompson, J. D., V. Prigent, and O. Poch (2004). LEON: multiple aLignment Evaluation Of Neighbours. ***Nucleic Acids Research***, **32**(4), pp. 1298–1307.

Thompson, J. D., J.-C. Thierry, and O. Poch (2003). RASCAL: rapid scanning and correction of multiple sequence alignments. ***Bioinformatics***, **19**(9), pp. 1155–1161.

Van Walle, I., I. Lasters, and L. Wyns (2005). SABmark: a benchmark for sequence alignment that covers the entire known fold space. ***Bioinformatics***, **21**(7), pp. 1267–1268.

Wallace, I. M., O. O'Sullivan, D. G. Higgins, and C. Notredame (2006). M-Coffee: combining multiple sequence alignment methods with T-Coffee. ***Nucleic Acids Research***, **34**(6), pp. 1692–1699.

Wang, L. and T. Jiang (1994). On the complexity of multiple sequence alignment. ***Journal of computational biology : a journal of computational molecular cell biology***, **1**(4), pp. 337–348.

Weinberger, K. Q. and L. K. Saul (2009). Distance metric learning for large margin nearest neighbor classification. ***Journal of Machine Learning Research***, **10**, pp. 207–244.

Wheeler, T. J. and J. D. Kececioglu (2007). Multiple alignment by aligning alignments. ***Proceedings of the 15th ISCB Conference on Intelligent Systems for Molecular Biology (ISMB), Bioinformatics***, **23**(13), pp. i559–i568.

Wheeler, T. J. and J. D. Kececioglu (2012). Opal: software for aligning multiple biological sequences (version 2.1.0). http://opal.cs.arizona.edu.

Wilbur, W. J. and D. J. Lipman (1983). Rapid similarity searches of nucleic acid and protein data banks. ***Proceedings of the National Academy of Sciences of the United States of America***, **80**, pp. 726–730.

Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. ***Biometrics Bulletin***, **1**(6), pp. 80–83.

Will, S., K. Reiche, I. L. Hofacker, P. F. Stadler, and R. Backofen (2007). Inferring Noncoding RNA Families and Classes by Means of Genome-Scale Structure-Based Clustering. ***PLoS Computational Biology***, **3**(4), pp. 680–691.

Woerner, A. and J. Kececioglu (2016). Faster metric-space nearest-neighbor search using dispersion trees. In preparation.

Wu, M., S. Chatterji, and J. A. Eisen (2012). Accounting for alignment uncertainty in phylogenomics. ***PLoS ONE***, **7**(1), pp. 1–10.

Yang, Z. (1993). Maximum-likelihood estimation of phylogeny from DNA sequences when substitution rates differ over sites. ***Molecular Biology and Evolution***, **10**(6), pp. 1396–1401.

Ye, Y., D. W.-l. Cheung, Y. Wang, S.-M. Yiu, Q. Zhang, T.-W. Lam, and H.-F. Ting (2015). `GLProbs`: Aligning multiple sequences adaptively. ***IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)***, **12**(1), pp. 67–78.

Zhihua, Z. (2012). ***Ensemble Methods: Foundations and Algorithms***. Chapman and Hall.