Estimating the Accuracy of Multiple Alignments and its Use in Parameter Advising[⋆]

Dan F. DeBlasio[1], Travis J. Wheeler[2], and John D. Kececioglu[1,⋆⋆]

[1] Department of Computer Science, The University of Arizona, USA
[2] Janelia Farm Research Campus, Howard Hughes Medical Institute, USA
kece@cs.arizona.edu

Abstract. We develop a novel and general approach to estimating the accuracy of protein multiple sequence alignments without knowledge of a reference alignment, and use our approach to address a new problem that we call *parameter advising*. 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. We evaluate this approach by applying it to the task of parameter advising: the problem of choosing alignment scoring parameters from a collection of parameter values to maximize the accuracy of a computed alignment. Our estimator, which we call `Facet` (for "feature-based accuracy estimator"), yields a parameter advisor that on the hardest benchmarks provides more than a 20% improvement in accuracy over the best default parameter choice, and outperforms the best prior approaches to selecting good alignments for parameter advising.

1 Introduction

Estimating the accuracy of a computed multiple sequence alignment without knowing the correct alignment is an important problem. A good accuracy estimator has broad utility, from building a meta-aligner that selects the best output of a collection of aligners, to boosting the accuracy of a single aligner by choosing the best values for alignment parameters. The accuracy of a computed alignment is typically determined with respect to a reference alignment, by measuring the fraction of substitutions in the core columns of the reference alignment that are present in the computed alignment. We estimate accuracy without knowing the reference by learning a function that combines several easily-computable features of an alignment into a single value.

In the literature, several approaches have been presented 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 of a computed alignment in recovering a correct alignment (where correctness is either with respect to the unknown structural alignment, as in our

[⋆] Research supported by US NSF Grant IIS-1050293 and DGE-0654435.
[⋆⋆] Corresponding author.

present study, or the unknown evolutionary alignment, as in simulation studies). One strategy is to develop a new *scoring function* on alignments that ideally is correlated with accuracy. These scoring functions combine local features of an alignment into a score, where the features typically include a measure of the conservation of amino acids in alignment columns. Scoring-function-based approaches include `AL2CO` by Pei and Grishin [9], `NorMD` by Thompson et al. [11], and `PredSP` by Ahola et al. [1]. The other general strategy 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 changing the input to a method, and (b) measure the *support* for the computed alignment among the collection [8,7,10,6]. 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. Support-based approaches include `MOS` by Lassmann and Sonnhammer [8], `HoT` by Landan and Graur [7], `GUIDANCE` by Penn et al. [10], and `PSAR` by Kim and Ma [6]. In our experiments, among scoring-function-based approaches, we compare just to `NorMD` and `PredSP`, since `AL2CO` is known [8] to be dominated by `NorMD`. Among support-based approaches, we compare just to `MOS` and `HoT`, as `GUIDANCE` requires alignments of at least four sequences (ruling out the many three-sequence protein reference aligments in suites such as `BENCH` [3]), while `PSAR` is not yet implemented for protein alignments.

The new approach we develop for accuracy estimation significantly improves on prior approaches, as we demonstrate through its performance on parameter advising. Compared to prior scoring-function-based approaches, we (a) introduce several *novel feature functions* that measure non-local properties of an alignment that show stronger correlation with accuracy, (b) consider *larger classes of estimators* beyond linear combinations of features, and (c) develop *new regression approaches* for learning a monotonic estimator from examples. Compared to support-based approaches, our estimator does not degrade on difficult alignment instances, where for parameter advising, good accuracy estimation is most needed. As shown in our advising experiments, support-based approaches lose the ability to detect accurate alignments of hard-to-align sequences, since most alternate alignments are poor and lend little support to the best alignment.

2 Accuracy Estimators

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 $[0, 1]$. We can then guarantee that the estimator has range $[0, 1]$ by ensuring that coefficients found by the training process yield a convex combination of features.

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 $1 \leq i \leq k$. Then our accuracy estimator $E(A)$ is a polynomial in the k variables $f_i(A)$. For example, for a degree-2 polynomial, $E(A) := a_0 + \sum_{1 \leq i \leq k} a_i \, f_i(A) + \sum_{1 \leq i,j \leq k} a_{ij} \, f_i(A) \, f_j(A)$. Learning an estimator from example alignments, as discussed in Section 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. The key is that such an estimator can always be reduced to the linear case by a change of feature functions. 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 $g_j(A) := \prod_{1 \leq i \leq k} \big(f_i(A)\big)^{p_i}$. Then 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 quickly becomes an issue. (Even a cubic estimator on 10 features already has 286 coefficients.) In our experiments, we focus on *linear* and *quadratic* estimators.

3 Learning the Estimator from Examples

To learn an accuracy estimator, we collect a training set of alignments whose true accuracy is known, and find coefficients for the estimator that give the best fit to true accuracy. We form the training set for our experiments by (1) collecting reference alignments from standard suites of benchmark protein multiple alignments; (2) for each such reference alignment, generating alternate alignments of its sequences by calling a multiple sequence aligner with differing values for the parameters of its alignment scoring function, in particular by varying its gap penalties, which can yield markedly different alignments; and (3) labeling each such alternate alignment by its accuracy with respect to the reference alignment for its sequences. We are careful to use suites of benchmarks for which the reference alignments are obtained by structural alignment of the proteins using their known three-dimensional structures. These alternate alignments together with their labeled accuracies form the *examples* in our training set.

Fitting to Accuracy Values. A natural fitting criterion 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^* that minimizes $\sum_{A \in \mathcal{S}} w_A \, \big| E_c(A) - F(A) \big|^p$, 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.) 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 quadratic program the linear inequalities $c \geq 0$ and $\sum_{0 \leq i < t} c_i \leq 1$.

When $p = 1$, the formulation corresponds to minimizing the L_1 norm. This is less sensitive to outliers than the L_2 norm, an advantage 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 inequalities $e_A \geq E_c(A) - F(A)$ and $e_A \geq F(A) - E_c(A)$, 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. We can also add inequalities that ensure E^* has range $[0, 1]$.

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. (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!) In this situation, when training set $\mathcal{S}$ is generated as described earlier, 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.

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 Section 5.) 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^* that minimize $\sum_{(A,B)\in\mathcal{P}} w_{AB}\, e_{AB}^p$, where e_{AB} is defined to be $\max\big\{\big(F(B)-F(A)\big) - \big(E_c(B)-E_c(A)\big),\, 0\big\}$, and 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 inequalities $e_{AB} \geq 0$ and $e_{AB} \geq \big(F(B) - F(A)\big) - \big(E_c(B) - E_c(A)\big)$, 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 select a sparse set $\mathcal{P}$ of pairs as follows. 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. A monotonic estimator, especially one that is used for parameter advising, should properly rank the set of alternate alignments for each benchmark. Consequently for each benchmark, we select all pairs of alternates whose difference in accuracy is at least ϵ, where ϵ is a tunable threshold. For the estimator to generalize outside 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 Pairs for Difference Fitting. 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 $\sum_{(A,B)\in\mathcal{P}:A\in\mathcal{B}} \frac{1}{2}\, w_{AB} + \sum_{(A,B)\in\mathcal{P}:B\in\mathcal{B}} \frac{1}{2}\, w_{AB} = 1$. 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. 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) 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. The proof is omitted due to page limits.

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, and runs in $O(k + m)$ time for k bins and m pairs in $\mathcal{P}$.*

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.

4 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 a dozen 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 reference alignments are structural alignments). Another aspect of the best alignment features is that they tend to use *nonlocal 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. 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. 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 Section 5 in the context of parameter advising.

The following are the feature functions we consider for our estimator.

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. (There are three basic types of *secondary structure* that a residue can have: α-helix, β-strand, and *coil*, which stands for "other.") 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` [4]. 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.) 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, blockiness measures the percentage of substitutions that are in α- or β-blocks.

While at first glance measuring blockiness might seem hard (as most combinatorial packing problems are in the worst-case), it can actually be computed in *linear time*, as the following theorem states. The key insight is that evaluating blockiness can be reduced to solving a longest path problem on a directed acyclic graph of linear size. The proof is omitted due to page limits.

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.*

Other Features. The remaining features are simpler than blockiness.

Secondary Structure Agreement. The secondary structure prediction tool `PSIPRED` [4] 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, normalized so they add up to 1, by $p_\alpha(i)$, $p_\beta(i)$, and $p_\gamma(i)$. Then we can estimate the probability that two residues i, j in a column have the same secondary structure type that is not coil by $P(i,j) := p_\alpha(i)\, p_\alpha(j) \,+\, p_\beta(i)\, p_\beta(j)$. To measure how strongly the secondary structure locally agrees around two residue positions, we compute a weighted average of P in a window of width $2\ell + 1$ centered at the positions: $Q(i,j) := \sum_{-\ell \le k \le \ell} w_k \, P(i{+}k, j{+}k)$, where the weights w_k form a discrete distribution that peaks at $k = 0$ (centered on i and j) and is symmetric. The value of the secondary structure agreement feature is then the average of $Q(i,j)$ over all residue pairs i, j in all columns.

Gap Features. A gap in a pairwise alignment is a maximal run of either insertions or deletions. We consider the following four features on gaps. In *Gap Coil Density*, for every pair of sequences, we measure the fraction of residues involved in gaps in the pairwise alignment induced by the sequence pair that are predicted by `PSIPRED` to be of secondary structure type *coil*, and average this measure over all induced pairwise alignments. *Gap Extension Density* counts the number of null characters in the alignment (the dashes that denote gaps), normalized by the total number of alignment entries. *Gap Open Density* counts the number of *runs* of null characters in the rows of the alignment, normalized by the total length of all such runs. For *Gap Compatibility*, as in cladistics, we encode the gapping pattern in the columns of an alignment by a binary state: residue, or null character. For an alignment in this encoding we then collapse together adjacent columns that have the same gapping pattern. We evaluate the 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. The feature measures the fraction of pairs of reduced columns that pass the test.

Conservation Features. We consider the following six features that measure conservation within alignment columns. In *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. The feature measures the fraction of encoded column pairs that pass the four-gametes test. We considered the standard reduced alphabets with 6, 10, 15, and 20 equivalency classes, and used the 10-class alphabet, which gave the strongest correlation with accuracy. *Amino Acid Identity* is usually called percent-identity. In each induced pairwise alignment, we measure the fraction of substitutions in which the residues have the same amino acid equivalency class. The feature averages this over all induced pairwise alignments. *Secondary Structure Identity* is like amino acid identity, except that instead of the protein's amino acid sequence, we use the secondary structure sequence predicted for the protein by `PSIPRED` [4] (a string over a 3-letter alphabet). *Average Substitution Score* computes the average score of all substitutions in the alignment using a `BLOSUM62` substitution-scoring matrix that has been shifted and scaled so the amino acid similarity scores are in the range $[0, 1]$. *Core Column Density* predicts core columns as those that do not contain null characters and whose fraction of pairs of residues that have the same amino acid equivalency class is above a threshold. The feature then normalizes the count of predicted core columns by the total number of columns in the alignment. *Information Content* measures the average entropy of the alignment, 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.

5 Application to Parameter Advising

In characterizing six stages in constructing a multiple sequence alignment, Wheeler and Kececioglu [12] 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 [12] 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. The preceding sections have addressed how to design estimator E; we now turn to how to find a good set P.

Finding an Optimal Parameter Set. Since the above advisor computes as many alignments as there are parameter choices in P, set P must be small for such an advisor to be practical. Given a bound on $|P|$, we would like to find a set P that, say, maximizes the true accuracy of the aligner $\mathcal{A}$ using the advisor's recommendation from P, averaged over a training collection of benchmarks S. Finding such an optimal set P is prohibitive, however, because the behavior of the advisor is an indirect function of the entire set P (through the estimator E), which effectively forces us to enumerate all possible sets P to find an optimal one.

Instead, we can directly find a set P that maximizes the true accuracy of aligner $\mathcal{A}$ using an *oracle* on P. The true accuracy of aligner $\mathcal{A}$ on a given input S, using an oracle on set P, is simply $\max_{p \in P} F\big(\mathcal{A}_p(S)\big)$, where again $F(A)$ is the true accuracy of alignment A. We can then find a set P^* that maximizes the average of the above quantity over all inputs S in a collection of benchmarks. We use this criterion, which seeks to maximize the performance of an *oracle* on the parameter set, for our definition of an optimal parameter set P.

We formulate this model as follows. Let U be the universe of parameter choices from which we want to find a parameter set $P \subseteq U$. For a parameter choice $p \in U$ and sequences S, denote the true accuracy of the aligner using p on S as $a_{p,S} := F\big(\mathcal{A}_p(S)\big)$. Given a bound k on the size of P, we want to find the optimal set $P^* := \operatorname{argmax}_{P \subseteq U} \sum_S \operatorname{argmax}_{p \in P} a_{p,S}$, where the summation is over all benchmarks S in a training set. (This is equivalent to maximizing the average true accuracy of the aligner over the training set using the oracle on P.) We call this problem, *Optimal Parameter Set.*

As might be expected, Optimal Parameter Set is NP-complete: it is equivalent to a classic problem from the field of operations research called *Facility Location.* Both problems can be tackled, however, by expressing them as an *integer linear program.* Page limits prevent us from giving the formulation of the integer linear program, but for a universe U of t parameter choices and a training set of n benchmarks, it has $O(tn)$ variables and $O(tn)$ constraints. For a large universe and a large training set, the resulting integer program can get very big.

In our computational experiments, we tackle the integer linear program as follows. First we solve the relaxed *linear program*, where the integer variables are allowed to take on *real values.* We then examine the optimal solution to this linear program, and for all the parameter choices p not chosen in the solution to the relaxed linear program, we *throw out* p from U, forming a new, reduced universe of parameter choices $\widetilde{U}$. Finally we solve the full integer linear program on the reduced universe $\widetilde{U}$. This approach, which we call solving the *reduced integer program*, is essentially a very accurate rounding heuristic for tackling the integer linear program. In our experiments, it is remarkably fast, allowing us to find high-quality solutions for very large instances with a universe of over 500 parameter choices and a training set of over 800 benchmarks.

6 Experimental Results

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 four estimators from the literature: `NorMD` [11], `MOS` [8], `HoT` [7], and `PredSP` [1]. (In terms of our earlier categorization of estimators, `NorMD` and `PredSP` are scoring-function-based, while `MOS` and `HoT` are support-based.) We refer to our estimator by the acronym `Facet` (short for "**F**eature-based **ac**curacy **e**s**t**imator").

In our experiments, for the collection of alignment benchmarks we used the `BENCH` suite of Edgar [3], which consists of 759 benchmarks, supplemented by a selection of 102 benchmarks from the `PALI` suite of Balaji et al. [2]. 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 each reference alignment in this benchmark collection, we generated alternate multiple alignments of the sequences in the reference using the multiple alignment tool Opal [12,13] with varying parameter choices. Each parameter choice is a vector of four integer values for scoring an alignment using affine gap penalties: a gap-open and gap-extension penalty for internal gaps, and the same for external gaps. These were selected from a universe U of 556 parameter choices [12]. (Set U was formed by finding a single optimal parameter choice through inverse parametric alignment [5], rounding this real-valued parameter choice to integers, and enumerating the Cartesian product of integer choices in the neighborhood of this central choice.) Opal also provides a default parameter choice from U that yields the highest average accuracy across the benchmarks.

For the experiments, we measure the *difficulty* of a benchmark S by the true accuracy of the alignment computed by Opal 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 $[0, .1]$ through $[.9, 1]$ are respectively: $13, 9, 24, 40, 30, 44, 66, 72, 129, 434$. To correct for this bias in oversampling of easy benchmarks, our approaches for learning an estimator nonuniformly weight the training examples, as described earlier.

To generate training and test sets for our parameter advising experiments on a given set $P \subseteq U$ of parameter choices, we used *three-fold cross validation*. For each bin, we evenly and randomly partitioned the benchmarks in the bin into three groups; we then formed three different splits of the entire set of benchmarks into a training class and a test class, where each split put one group in a bin into the test class and the other two groups in the bin into the training class; finally, for each split we generated a test set and a training set of example alignments by generating $|P|$ alignments from each benchmark S in a training or test class by running Opal on S using each parameter choice in P. An estimator learned on the examples in the training set was evaluated on the examples in the associated test set. The results that we report are averages over three runs, where each run is on one of these training and test set pairs. For a set P of 10 parameters, each run has over $5,700$ training examples and $2,800$ test examples.

Selecting Features. Of the features listed in Section 4, not all are equally informative, and some can weaken an estimator. We selected subsets of features for use in accuracy estimators as follows. For each of the twelve features, we first learned an estimator that used that feature alone, and ranked all twelve of these single-feature estimators according to the average accuracy across the difficulty bins of the resulting parameter advisor using that estimator. Then a greedy procedure was used to find a subset of the features, considering them in this order, whose fitted estimator gave an advisor with the highest average accuracy. This procedure was separately run to find good feature subsets for the optimal parameter sets on 5, 10, and 15 parameters. To find a good *overall* feature set that works well for differing numbers of parameters, we examined all subsets of features considered during runs of the greedy procedure, and chose the subset that had the highest accuracy averaged across the 5-, 10-, and 15-parameter sets.

The best overall feature set found by this process is a 5-feature subset consisting of the following features: secondary structure blockiness (`BL`), secondary structure agreement (`SA`), secondary structure identity (`SI`), substitution compatibility (`SP`), and average substitution score (`AS`). The corresponding fitted estimator is $(.174)\,\texttt{SA} + (.172)\,\texttt{BL} + (.168)\,\texttt{SI} + (.167)\,\texttt{SP} + (.152)\,\texttt{AS} + (.167)$. This 5-feature estimator, when advising using the optimal 10-parameter set, has average accuracy 57.9%. By comparison, the 12-feature estimator has corresponding average accuracy 56.3%. Clearly feature selection helps.

To give a sense of the impact of alternate fitting approaches and higher-degree polynomials, for this 5-feature set, the average accuracy on the 10-parameter set of the linear estimator with difference fitting is 57.9%, and with value fitting is 55.9%; the quadratic estimator with difference fitting is also 57.9%, and with value fitting is 55.8%. In general, we experience only marginal improvement with a quadratic estimator over a linear estimator, even using regularization, though difference fitting does provide a significant improvement over value fitting.

Comparing Estimators to True Accuracy. To examine the fit of an estimator to true accuracy, the scatter plots in Fig. 1 show the value of the `Facet`, `MOS`, and `PredSP` estimators on all example alignments in the 15-parameter test set.

The ideal estimator would be monotonic increasing in true accuracy. Comparing the scatter plots, `MOS` has the highest slope and spread, `PredSP` has the lowest slope and spread, while `Facet` has intermediate slope and spread. This better compromise between slope and spread may be what leads to improved performance for `Facet`.

Performance on Parameter Advising. To evaluate these methods for parameter advising, we found parameter sets P using our approach that solves the reduced integer linear program, for $|P| = 1, \ldots, 15$. We show results for the resulting parameter advisors that use `Facet`, `MOS`, `PredSP`, `HoT`, and `NorMD`. Each advisor is used within `Opal` to compute an alignment. Advisors are compared by the resulting true accuracy of `Opal`, first averaged within each bin and then averaged across all bins. We also do the same comparison looking at the rank of the alignment chosen by the advisor, where the alignments generated by the $|P|$ parameter choices are ordered by their true accuracy.

Figure 2 shows the performance of the four best resulting advising methods on the optimal set P of 10 parameters, with respect to accuracy of the alternate alignment chosen by the advisor, averaged over the benchmarks in each difficulty bin. The oracle curve shows what would be achieved by a perfect advisor. The figure also shows the expected performance of a purely *random advisor*, which simply picks a parameter from P uniformly at random. In the figure, note that the advisor that uses `Facet` is the only one that always performs at or above random in accuracy. The `Facet` advisor is also strictly better than every other advisor in accuracy on all but one bin, at difficulty $[.6, .7]$. As the figure shows, `Facet` gives the most help in the hardest bins.

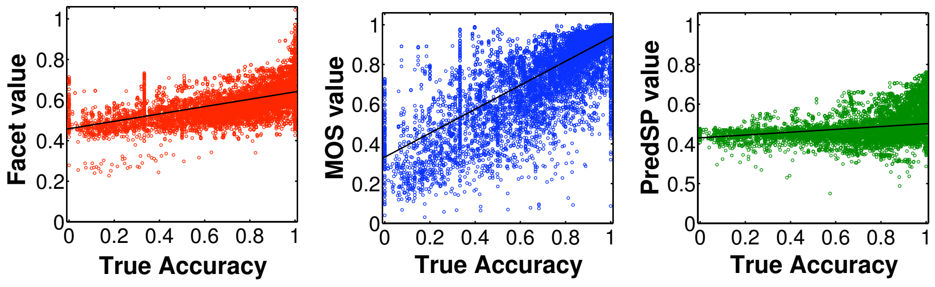

Fig. 1. Correlation of estimators with accuracy. Each scatter plot shows the value of an estimator versus true accuracy for all alignments in the 15-parameter test set.

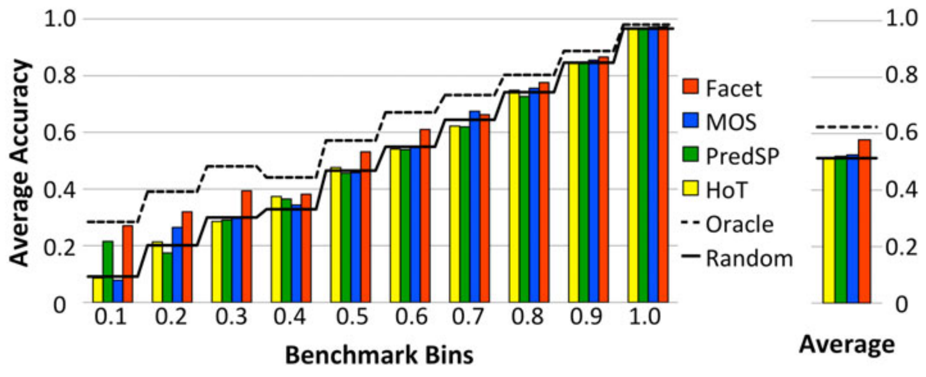

Fig. 2. Advising accuracy across benchmark bins. Each bar shows an estimator's performance in advising, averaged over benchmarks in the bin, for the 10-parameter set.

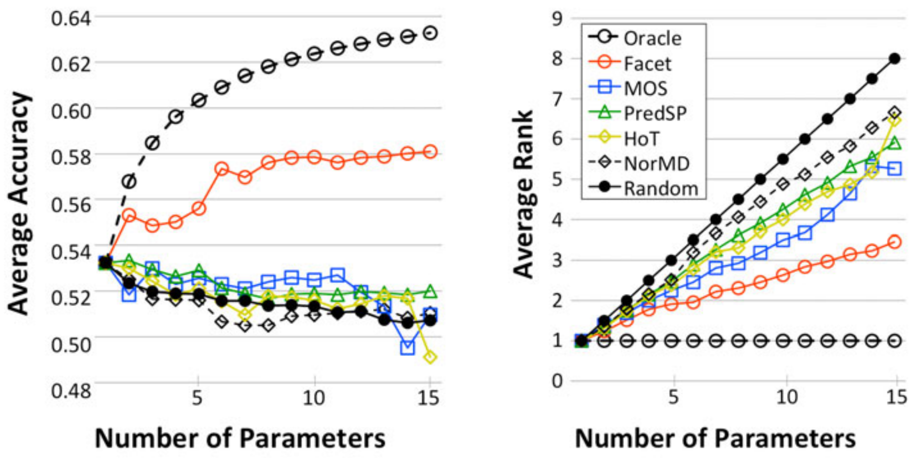

Fig. 3. Advising accuracy and rank for varying numbers of parameters. At each number of parameters, the curves show accuracy or rank averaged over all benchmark bins.

Figure 3 shows the advising performance of the estimators, uniformly averaged across all bins, as a function of the number of parameter choices $k = |P|$ for $k = 1, \ldots, 15$. Notice that with respect to accuracy, the performance of the `MOS`, `PredSP`, `HoT`, and `NorMD` advisors tends to decline as the number of parameter choices increases (and for some is not far from random). The `Facet` advisor generally improves with more parameters, though beyond 10 parameters the improvement is negligible. We remark that the performance of the oracle, if its curve is continued indefinitely to the right, reaches a limiting accuracy of 75.3%.

7 Conclusion

We have presented an efficiently-computable estimator for the accuracy of protein multiple alignments. The estimator is a polynomial function of alignment features whose coefficients are learned from example alignments using linear and quadratic programming. We evaluated our estimator in the context of parameter advising, and show it consistently outperforms other approaches to estimating accuracy when used for advising. Compared to using the best default parameter choice, the resulting parameter advisor using a set of 10 parameters provides a 20% improvement in multiple alignment accuracy on the hardest benchmarks.

References

1. Ahola, V., Aittokallio, T., Vihinen, M., Uusipaikka, E.: Model-based prediction of sequence alignment quality. Bioinformatics 24(19), 2165–2171 (2008)
2. Balaji, S., Sujatha, S., Kumar, S.S.C., Srinivasan, N.: PALI: a database of alignments and phylogeny of homologous protein structures. NAR 29(1), 61–65 (2001)
3. Edgar, R.C.: `http://www.drive5.com/bench` (2009)
4. Jones, D.T.: Protein secondary structure prediction based on position-specific scoring matrices. Journal of Molecular Biology 292, 195–202 (1999)
5. Kim, E., Kececioglu, J.: Learning scoring schemes for sequence alignment from partial examples. IEEE/ACM Trans. Comp. Biol. Bioinf. 5(4), 546–556 (2008)
6. Kim, J., Ma, J.: PSAR: measuring multiple sequence alignment reliability by probabilistic sampling. Nucleic Acids Research 39(15), 6359–6368 (2011)
7. Landan, G., Graur, D.: Heads or tails: a simple reliability check for multiple sequence alignments. Molecular Biology and Evolution 24(6), 1380–1383 (2007)
8. Lassmann, T., Sonnhammer, E.L.L.: Automatic assessment of alignment quality. Nucleic Acids Research 33(22), 7120–7128 (2005)
9. Pei, J., Grishin, N.V.: AL2CO: calculation of positional conservation in a protein sequence alignment. Bioinformatics 17(8), 700–712 (2001)
10. Penn, O., Privman, E., Landan, G., Graur, D., Pupko, T.: An alignment confidence score capturing robustness to guide tree uncertainty. MBE 27(8), 1759–1767 (2010)
11. Thompson, J.D., Plewniak, F., Ripp, R., Thierry, J.C., Poch, O.: Towards a reliable objective function for multiple sequence alignments. JMB 314, 937–951 (2001)
12. Wheeler, T.J., Kececioglu, J.D.: Multiple alignment by aligning alignments. Bioinformatics 23, i559–i568 (2007); Proceedings of the 15th ISMB
13. Wheeler, T.J., Kececioglu, J.D.: Opal: software for aligning multiple biological sequences. Version 2.1.0 (January 2012), `http://opal.cs.arizona.edu`