TF-MoDISco v0.4.4.2-alpha: Technical Note

Avanti Shrikumar
Stanford University
avanti@stanford.edu

Katherine Tian
The Harker School
19katherinet@students.harker.org

Anna Shcherbina
Stanford University
annashch@stanford.edu

Žiga Avsec
Technical University of Munich
zigaavsec@gmail.com

Abhimanyu Banerjee
Stanford University
manyu@stanford.edu

Mahfuza Sharmin
Stanford University
msharmin@stanford.edu

Surag Nair
Stanford University
surag@stanford.edu

Anshul Kundaje
Stanford University
akundaje@stanford.edu

Abstract

TF-MoDISco (Transcription Factor Motif Discovery from Importance Scores) is an algorithm for identifying motifs in basepair-level importance scores computed on genomic sequences. This paper describes the methods behind TF-MoDISco v0.4.4.2-alpha (`https://github.com/kundajelab/tfmodisco/tree/v0.4.2.2-alpha`).

1 Introduction

Convolutional neural networks have been used in recent years to successful learn regulatory patterns in genomic DNA [1] [3] [4]. Combinations of Transcription Factors (TFs) bind combinations of motifs in DNA sequence at non-coding regulatory elements to control gene expression. While the core sequence motifs of a subset of TFs are relatively well-known, the role of flanking nucleotides that influence *in vivo* TF binding, as well as the combinatorial interactions with other TFs remain largely uncharted. Deep learning models are appealing for this problem because of their ability to learn complex, hierarchical, predictive patterns directly from raw DNA sequence, thus removing the need to explicitly featurize the data (such as featurization using a database of known motifs). Convolutional Neural Networks (CNNs) in particular contain several hierarchical layers of pattern-matching units referred to as convolutional filters that are well suited to learning from DNA sequence. In these models, each convolutional filter learns a sequence pattern that is analogous to a position weight matrix (PWM). The filter is scanned (convolved) with the sequence to produce a score for the strength of the match at each position. Later convolutional layers operate on the scores from all filters in the previous layer, allowing the net to learn complex higher-order patterns.

A barrier to the adoption of deep learning models for genomic applications is the difficulty in interpreting the models. While several methods such as [4, 7, 8] have been developed to assign importance scores to each base of an input sequence, methods for learning re-occurring patterns are largely limited to variations of visualizing the learned representations of individual CNN filters [4, 3, 1]. In practice, this is problematic because CNNs learn highly distributed representations, meaning that the patterns found by individual convolutional neurons may not be very informative.

Here, we describe version 0.4.2.2-alpha of TF-MoDISco, a novel method for identification of high-quality, consolidated motifs using deep learning. The critical insight of TF-MoDISco is that importance scores on the inputs are computed using information from all the neurons in the network; thus, by focusing on segments of high importance in the inputs and clustering the segments, it should be possible to identify consolidated motifs learned by the deep learning model. The implementation is available at `https://github.com/kundajelab/tfmodisco/tree/v0.4.2.2-alpha`.

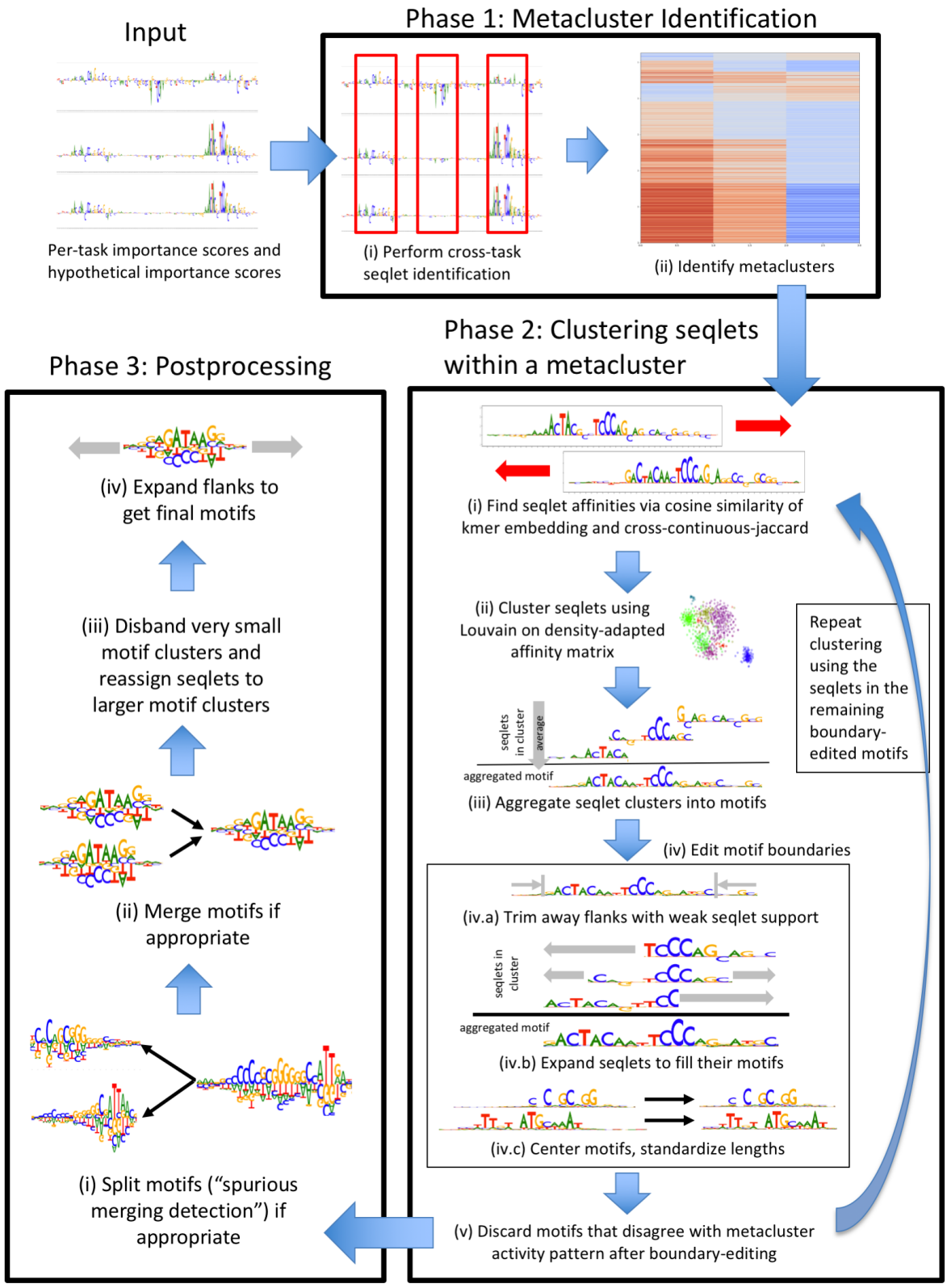

Figure 1: **Summary of TF-MoDISco**

2 Input to TF-MoDISco

TF-MoDISco takes as its input per-base importance scores for every task. These importance scores can be derived through a variety of methods, including the ones described in DeepLIFT[7]. A positive importance value for a particular task indicates that the base influences the output of the network towards a positive prediction for the task, and a negative importance indicates that the base influences the output of the network towards a negative prediction. Scores that are near zero indicate that the particular base is unimportant for the task in question.

We found that TF-MoDISco results were better if, in addition to using importance scores on the input sequence, we incorporated information about *hypothetical* importance if other bases were present. A hypothetical importance score answers the question "if I were to mutate the sequence at a particular position to a different letter, what would the importance on the newly-introduced letter look like?". As a specific example, consider a basic importance-scoring method such as gradient $\times$ input. When a sequence is one-hot encoded (i.e. 'input' can be either 1 or 0), the value of gradient $\times$ input would be zero on all bases that are absent from the sequence and equal to the gradient on the bases that are present (here, 'base' refers to a specific letter at a specific position; at every position, only one of ACGT can be present). If a single position were mutated to a different base, one might anticipate that the value of gradient $\times$ input at the newly-introduced base would be close to the current value of the gradient (assuming that the gradient doesn't change dramatically as a result of the mutation). Thus, the gradients give a readout of what the contributions to the network would be if different bases were present at particular positions; if (gradient $\times$ input) is used as the importance scoring method, then the gradients alone would be a good choice for the "hypothetical importance". In practice, the "hypothetical importance" behaves like an autocomplete of the sequence, giving insight into what patterns the net was looking for at a given region (Fig. 2).

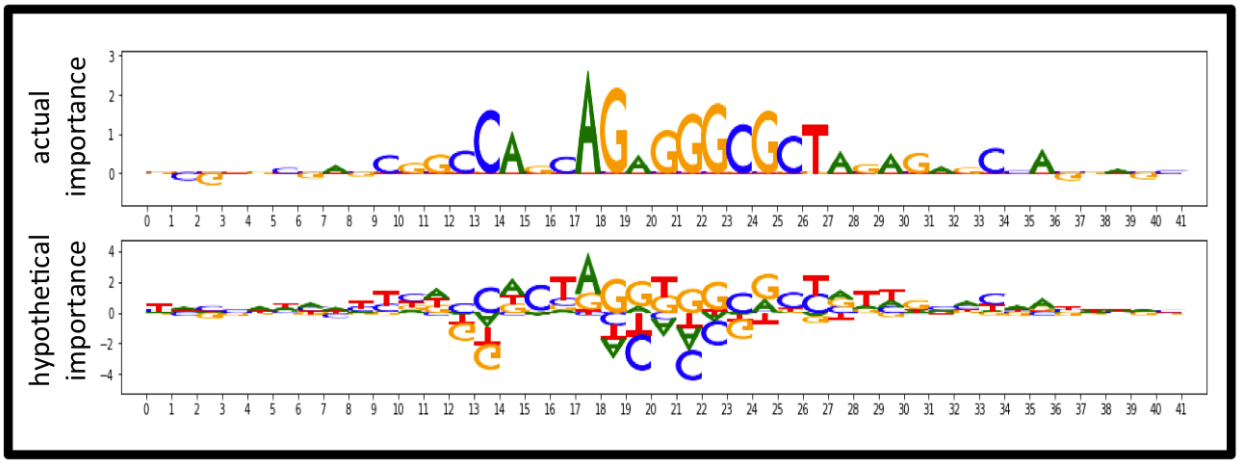

Figure 2: **Actual and hypothetical importance for CTCF task on an example sequence containing the CTCF motif**. The hypothetical importance reveals the impact of other bases not present in the original sequence.

What is a good choice of hypothetical importance when DeepLIFT scores are used? DeepLIFT defined terms called the multipliers $m_{\Delta x \Delta t}$ such that:

$$m_{\Delta x \Delta t} = \frac{C_{\Delta x \Delta t}}{\Delta x}$$

Where x is the input, t is the output that we are computing contributions to, Δx is the change in x (formally the 'difference-from-reference') and $C_{\Delta x \Delta t}$ is the contribution of change in x to the change in t. If x^h denotes a hypothetical value of the input, then we can approximate $C_{\Delta x^h \Delta t}$ as:

$$C_{\Delta x^h \Delta t} = \Delta x^h m_{\Delta x^h \Delta t} \quad (1)$$

In other words, we use the multipliers on the current input, but substitute the difference-from-reference of the hypothetical value of the input. Note that if the reference is set to 0 and $m_{\Delta x \Delta t}$ is taken to be

the gradient of t w.r.t. x, then the formula reduces to the case of gradient $\times$ input described in the previous paragraph. When the reference is not 0 and the input is subject to a one-hot encoding constraint, as is the case with DNA sequence input, care must be taken to project the contributions from the difference-from-reference of all the bases at a position onto the base that is actually present. A jupyter notebook with code illustrating how to obtain importance scores and hypothethetical scores using DeepLIFT on genomic sequences is avaiable at `https://github.com/kundajelab/tfmodisco/blob/v0.4.2.2-alpha/examples/simulated_tf_binding/Generate%20Importance%20Scores.ipynb`.

In future versions of TF-MoDISco, we envision allowing the user to supply other kinds of data tracks for TF-MoDISco clustering, such as the activations or importance scores on some intermediate convolutional layer, methylation/accessibility information, etc.

3 TF-MoDISco, Phase 1: Metaclusters

The goal of the first phase is to identify segments of the input, termed "seqlets', that have substantial contribution to one or more of the output tasks, and to then cluster these seqlets into 'metaclusters' that each have a distinct pattern of contribution to the various tasks.

3.1 Seqlet identification

After importance scores are obtained, the next step is to identify portions of the input, referred to as "seqlets", that have substantial contribution to one or more tasks. The approach taken is to first obtain a list of seqlet locations for each individual task and to then unify the lists across tasks. Seqlet identification for each task proceeds as follows:

- For each task, the importance scores are summed in sliding windows of core size `sliding_window_size`.
- The mode is identified, and the scores are centered around the mode by subtracting the mode. A one-tailed laplace distribution is fit to the 5th percentile of positive scores, and another one-tailed laplace distribution is fit to the 5th percentile of the absolute value of the negative scores. These laplace distributions are used to estimate False Discovery Rates under the assumption that scores with low magnitudes are mostly noise. We use different laplace distributions for positive and negative scores in case the positive and negative noise are asymmetric in their distributions.
- Based on the laplace distribution, FDRs are estimated for each value as [expected number of noise hits above value]/[actual number of hits above value]. The threshold corresponding to a target FDR of `target_seqlet_fdr` is identified. If no such threshold exists, the most extreme value in the distribution is used as the threshold.
- If the total number of windows with a sum that passes the positive and negative thresholds is less than `min_seqlets_per_task` $\times 0.5 \times$ `sliding_window_size`, the positive and negative thresholds are recalculated by sorting the windows in descending order of absolute value and picking a threshold such that the number of passing windows (in terms of absolute value) is `min_seqlets_per_task` $\times$ $0.5 \times$ `sliding_window_size`. The negative threshold is then set to be the minus of the positive threshold. Note that the actual number of seqlets may be smaller than `min_seqlets_per_task` after overlapping windows are removed. In later versions, we will adjust the algorithm such that the minimum number of seqlets obtained after overlap filtering is exactly `min_seqlets_per_task`.
- After the final threshold is determined, all windows with a sum that does not pass the threshold are filtered out.
- The following process is then repeated for each sequence:
  - Identify the window containing a total importance score of the highest magnitude
  - Expand the window on either side by flank size `flank_size`. These coordinates will be used to make a seqlet.
  - Filter out all windows that would overlap this by over 50% (after flank expansion)

The lists of seqlets across all tasks are then unionized. If a pair of seqlets overlaps by more than `overlap_portion` (which defaults to 50%), the seqlet with the higher score for its respective task is retained; as seqlet flanks are expanded in later steps of the algorithm, there is not a great concern of losing information from discarding overlapping seqlets.

3.2 Metaclustering

After a unified list of seqlets is obtained across all tasks, seqlets are clustered into metaclusters according to their contribution scores across tasks. The contribution of a seqlet to a particular task is taken to be the total per-position contribution score for that task in the central `sliding_window_size` basepairs of the seqlet. However, before we can cluster these scores across tasks, it is necessary to transform the scores such that scores from different tasks are comparable (by this, we mean that the scores for different tasks can be on different scales depending on the confidence of the model in that particular task, so some kind of normalization is needed). We do this transformation using the Laplace distributions that were calculated in **Sec. 3.1**; we replace each score with the CDF of the corresponding one-tailed Laplace distribution and apply a negative sign if the original score was negative (so, an extreme positive score would get transformed to +1.0, and an extreme negative score would get transformed to -1.0). Note: We have observed that, depending on the dataset, the Laplace distribution may not be a good fit for the distribution of seqlets scores. Future iterations of TF-MoDISco may replace the Laplace CDF with the percentile of the seqlet score in the case that the Laplace distribution is a poor fit.

We now describe the metaclustering algorithm in detail. We introduce the concept of an *activity pattern*, which is a vector where each element is a 1, 0 or a -1. An activity pattern acts as a coarse-grained summary of a seqlet's activity across all the tasks. If there are n tasks, then there are n^3 possible activity patterns. The goal of our metaclustering algorithm is to assign each seqlet to a distinct activity pattern, which will represent the metacluster.

Let v' denote the vector containing the transformed scores for all tasks for some seqlet. We say that v' is *strongly compatible* with an activity pattern p if, for every corresponding element v'_i and p_i, either $v'_i p_i >$ `strong_threshold` or $p_i = 0$ (that is, either $|v'_i|$ should exceed the `strong_threshold` in the direction indicated by p_i, or p_i should be 0; 0 is best understood as "no constraint" on a task). Similarly, we say that v' is *weakly compatible* with an activity pattern p if, for every corresponding element v'_i and p_i, either $v'_i p_i >$ `weak_threshold` or $p_i = 0$. Note that the all-zeros activity pattern is compatible with all seqlets by both strong and weak thresholds.

How are `strong_threshold` and `weak_threshold` set? `strong_threshold` is taken to be the most lenient CDF cutoff used across all tasks during per-task seqlet identification; the goal is to ensure that every seqlet has a transformed score of absolute value greater than `strong_threshold` for at least one task. The `weak_threshold` is defined by the user, but is set to be equal to `strong_threshold` if the user specifies a `weak_threshold` that exceeds `strong_threshold`.

The metaclustering algorithm then proceeds as follows:

- For each possible activity pattern, compute the number of "strongly compatible" seqlets. Note that a single seqlet is compatible with multiple activity patterns as some activity patterns are 0 for particular tasks.
- Filter out activity patterns that have fewer than `min_metacluster_size` compatible seqlets.
- Map each seqlet to the most specific activity pattern (that is, the activity pattern with the fewest zeros) with which it is weakly compatible. If a seqlet is compatible with two activity patterns of equal specificity, it is assigned to the one for which it has the higher absolute transformed score.

Emprically, we found that this metaclustering strategy produced metaclusters with relatively clean and distinct patterns relative to strategies like k-means clustering or community detection. In future versions of TF-MoDISco, we anticipate allowing the user to provide their own metaclusters that can supersede or act in combination with this metaclustering strategy. These metcalusters could be based on numerous alternative features such as the activations of the last hidden layer, the output label vector, or orthogonal signals such as methyation levels.

4 TF-MoDISco, Phase 2: Clustering Within a Metacluster

In Phase 2 of TF-MoDISco, we perform subclustering of the seqlets in a metacluster using the importance scores for the subset of tasks that are relevant to the metacluster. We consider a task to be relevant to a metacluster if the activity pattern of the metacluster is nonzero for that task.

4.1 Affinity Matrix Computation

The fist step of Phase 2 is to compute affinities between every pair of seqlets, so that these affinities can be later clustered. Because this step scales with $O(N^2)$, where N is the number of seqlets, we perform the computation in two steps: first, we compute a coarse-grained affinity matrix using a relatively quick method, and then for each node, we compute affinities for `nearest_neighbors_to_compute` nearest neighbors (as determined by the coarse-grained affinity matrix) using a more sophisticated but also more time-consuming method. The default value of `nearest_neighbors_to_compute` is 500.

Coarse-Grained Affinity Matrix Computation: Gapped k-mer embedding

For the coarse-grained affinity matrix, a gapped k-mer embedding is derived for each seqlet, and the cosine similarity between seqlet embeddings is used as the affinity. Note that this aspect of the workflow is highly specific to DNA sequence, and would have to be replaced with another way of computing affinity matrices if the user wishes to cluster patterns that are not DNA sequence (we may add support for this in future versions of TF-MoDISco). The gapped k-mer embedding proceeds as follows:

- For each seqlet, a single hypothetical contributions track is created by taking the sum of the hypothetical contributions for the tasks that are relevant for the metacluster's activity pattern (a task is "relevant" if it is nonzero in the activity pattern). If a task is negative in the activity pattern, the hypothetical contributions are multiplied by -1 before being added to the sum.
- Gapped k-mers (potentially allowing for mismatches) are identified in the seqlet by scanning the one-hot encoded sequence.
- For each gapped k-mer match, the "score" of the match is taken to be the dot product of the one-hot encoded representation of the gapped k-mer and the portion of the summed hypothetical contributions track that it overlaps.
- A total score for each gapped kmer is computed by summing the scores over all matches for that gapped kmer in the seqlet. This vector of total scores serves as the embedding.

For efficiency, the gapped kmer scoring is implemented using convolutional operation on the GPU.

Fine-grained Affinity Matrix Computation: The Continuous Jaccard Similarity

For the `nearest_neighbors_to_compute` nearest neighbors of each seqlet as computed by the coarse-grained affinity matrix, we compute affinities using a more sophisticated (but computationally slower) similarity metric. Initially, we cross-correlated the importance score tracks of two seqlets and used the correlation at the best alignment as the similarity. However, using cross-correlation in this context has a drawback, which we explain here.

Consider the following toy example: we have a vector $v_1 = (-1, -1, -2, 4, -1, -1, -1)$ of scores, and we are comparing it to two other vectors: $v_2 = (0, 0, 0, 4, 0, 0, 0)$ and $v_3 = (-1, -1, -2, 0, -1, -1, -1)$. The pearson correlation of v_1 with v_2 is equal to the cosine similarity between v_1 and v_2 after mean-normalization, and is 0.98. By contrast, the cosine similarity between v_1 and v_3 is 0.87.

To appreciate why the correlation between v_1 and v_2 is higher than the correlation between v_1 and v_3 even though v_1 and v_2 agree on the value of only one element while v_1 and v_3 agree on the value of 6, note that the formula for the correlation involves taking the elementwise product of terms. Thus, the formula has a polynomial degree of two, meaning that values of larger magnitude will have a disproportionately large influence on the correlation compared to values of smaller magnitude. In the context of comparing seqlets, this can be problematic because the importance scores at individual seqlets can be quite noisy, so a similarity metric that is highly sensitive to the magnitudes of the scores can produce unintuitive results.

As an alternative to correlation, we propose the following similarity measure, inspired by the Jaccard similarity. First, we define a notion of 'intersection' and 'union' for real numbers as follows:

$$\begin{aligned} x \cap y &= \min(|x|, |y|) \times \text{sign}(x) \times \text{sign}(y) \\ x \cup y &= \max(|x|, |y|) \end{aligned} \tag{2}$$

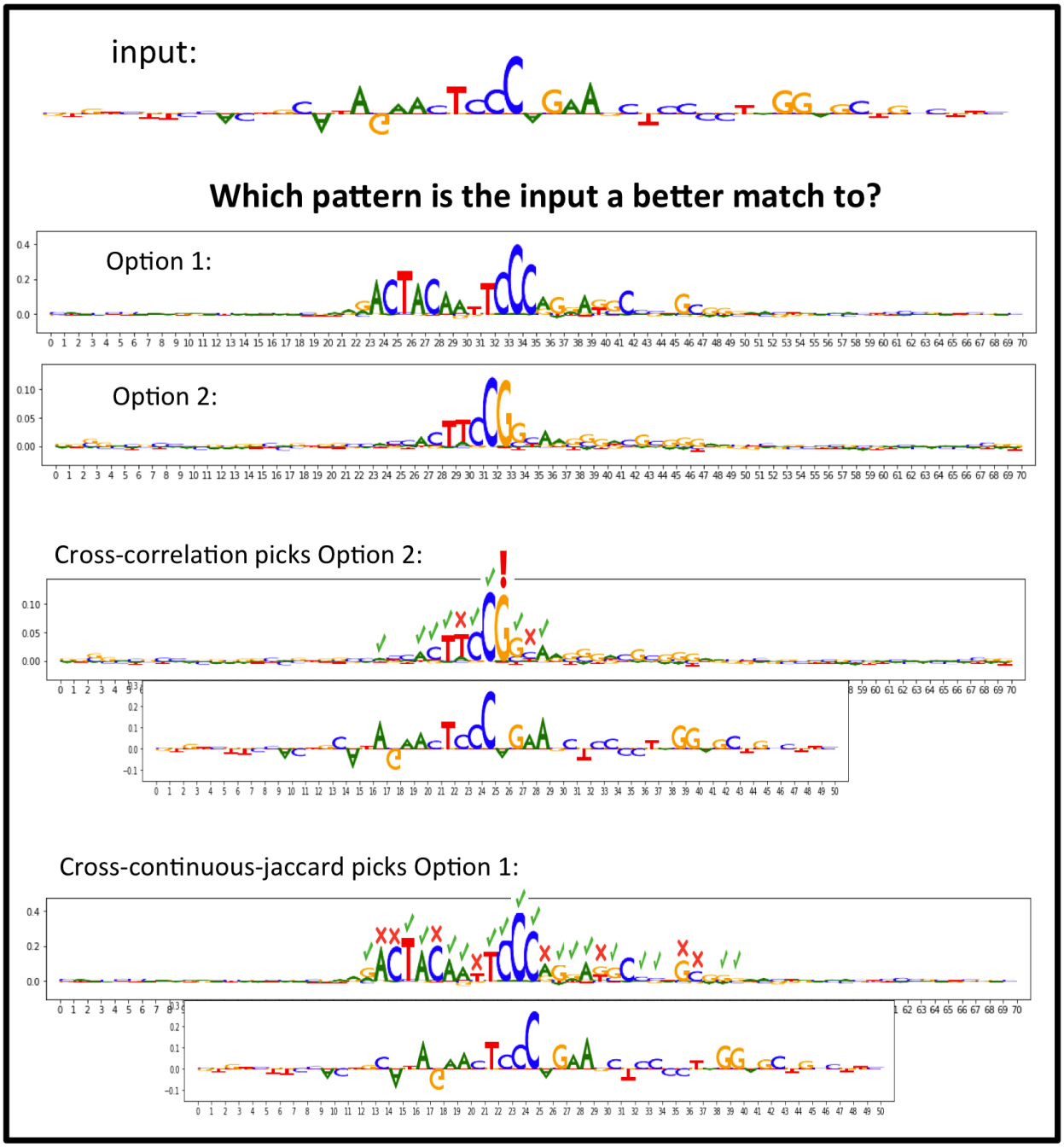

Figure 3: **Continuous Jaccard similarity is preferable to cross-correlation for matching seqlets**. Green checkmarks indicate matching positions.

Note that $x \cap y$ will have a positive sign if the signs of x and y agree, and a negative sign otherwise. We define the continuous Jaccard similarity between v_1 and v_2 as:

$$\text{ContinuousJaccard}(v_1, v_2) = \frac{\sum_i v_1^i \cap v_2^i}{\sum_i v_1^i \cup v_2^i} \tag{3}$$

Here, v_1^i denotes the ith element of v_1. The Continuous Jaccard similarity will have a value of -1 if $v_1 = -v_2$, and a value of 1 if $v_1 = v_2$. Returning to our toy example, we find that $(-1, -1, -2, 4, -1, -1, -1)$ has a Continuous Jaccard similarity of $\frac{4}{11}$ with $(0, 0, 0, 4, 0, 0, 0)$ and $\frac{7}{11}$ with $(-1, -1, -2, 0, -1, -1, -1)$.

As an illustration of the effectiveness of our similarity metric in practice, consider the example in Fig. 3. The Continuous Jaccard similarity matches the input scores to the pattern that agrees with it at more positions compared to the match produced by using correlation.

What do we use as the input to our continuous jaccard similarity when we compare two seqlets? We use a combination of the hypothetical importances and the actual importances for all tasks deemed to be relevant for the seqlets. Specifically, let $H^t_{s_1}$ denote the hypothetical importance scores for seqlet s_1 and task t, and let $I^t_{s_1}$ denote the actual importance scores for seqlet s_1 and task t. Both $H^t_{s_1}$ and $I^t_{s_1}$ have dimensions $l_s \times 4$, where l_{s_1} is the length of a seqlet. Let $\hat{H}^t_{s_1}$ and $\hat{I}'^t_{s_1}$ denote L1-normalized version of $H^t_{s_1}$ and $I^t_{s_1}$ (in other words, the sum of the absolute values of the elements in each of $\hat{H}^t_{s_1}$ or $\hat{I}'^t_{s_1}$ totals 1). We form a new matrix S_1 by concatenating the matrices $\hat{H}^t_{s_1}$ and $\hat{I}'^t_{s_1}$ for all relevant tasks along the second dimension. If m is the total number of relevant tasks, the final dimensions of S_1 will be $l_s \times 4 \times 2 \times m$ (the factor of 2 is introduced because we are using both actual importances and hypothetical importances for every task).

The matrices S_1 and S_2 are then padded with zeros and compared at all alignments where the non-padded matrices overlap at at least `min_overlap_while_sliding`l_s positions using the Continuous Jaccard similarity metric (the default value of `min_overlap_while_sliding` is 0.7). The maximum Continuous Jaccard similarity over all possible alignments is take to be the similarity between s_1 and s_2.

Noise filtering

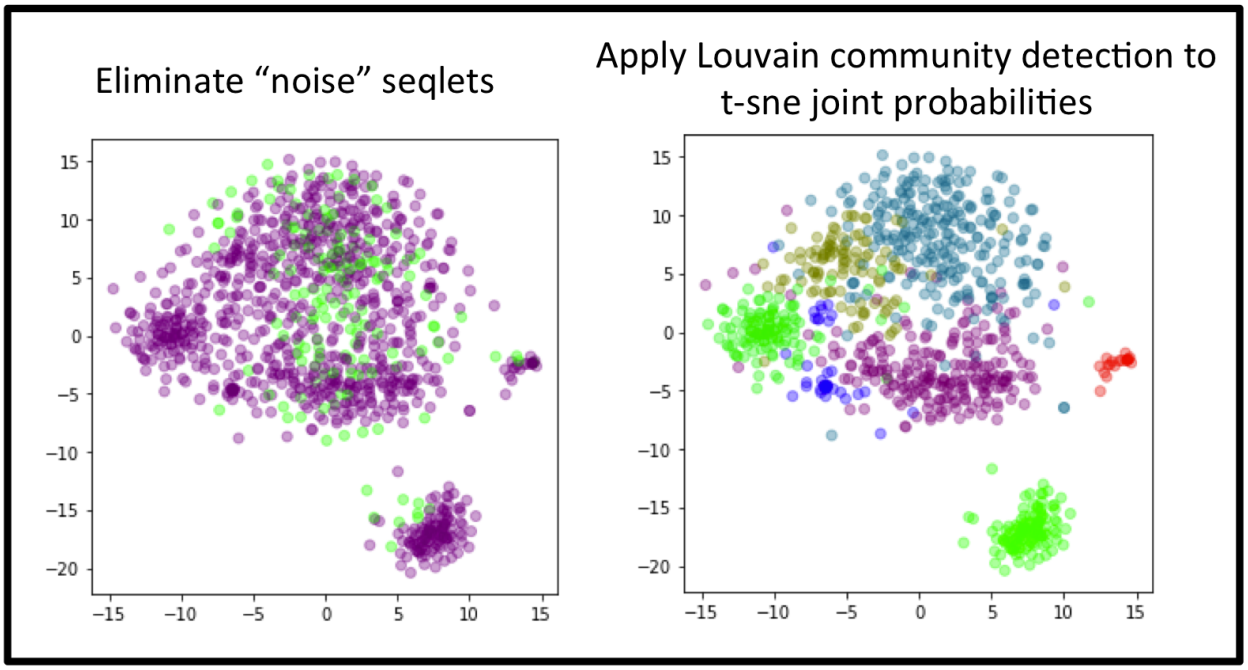

Figure 4: **Noise filtering and seqlet clustering.** Left: Seqlets with poor spearman correlation between the coarse-grained and fine-grained affinities were discarded. Right: The remain seqlets were clustered to form an initial set of motifs.

We found that seqlets which had very poor correlation between the coarse-grained affinities and the fine-grained affinities tended to resemble noise. We leverage this observation and discard all seqlets for which the spearman correlation between coarse-grained and fine-grained affinities is below `affmat_correlation_threshold`, which defaults to 0.15. See the left panel of Fig. 4 for a tsne-embedding showing the seqlets eliminated by our noise filtering step.

4.2 Clustering the Affinity Matrix

Now that we have an affinity matrix, the next step is to obtain clusters. We first perform two successive transformations of the affinity matrix: the first, inspired by t-sne, is intended to adapt the notion of distance to the local density of the data, and the second is done to reduce the instability of Louvain community detection.

Transformation 1: Adapting the Notion of Distance to the Local Density of the Data

A challenge with using the original affinity matrix is that the notion of what is 'close' can vary with the motif cluster. For example, a weak motif might have more weak matches present in the data compared to the strongest motif, so what we might consider a good similarity score for the weak motif may not be appropriate for the strong motif. One way to address this is to adapt the notion of 'close' to the local density of the data; thus, if a seqlet has a lot of neighbors with high similarity, we can afford to be more stringent in our definition of 'close'. By contrast, if a seqlet's nearest neighbors have relatively low similarity, we may have to be more lenient.

To achieve this, we borrow from t-sne [6]. The first step of t-sne is to compute a matrix of conditional probabilities $p_{j|i}$ that represent the likelihood that seqlet s_j is picked conditioned on s_i having already been picked, where $p_{j|i}$ is proportional to a Gaussian centered on i:

$$p_{j|i} \propto \exp\left(-\beta_i d_{s_i s_j}\right) \tag{4}$$

Where $d_{s_i s_j}$ is the distance between seqlet s_i and s_j. Crucially, the value of β_i is tuned such that the $p_{j|i}$ distribution achieves a target perplexity. A distribution in which the k nearest seqlets of i each had probability $\frac{1}{k}$ and every other seqlet had probability zero would have a perplexity of exactly k. Thus, perplexity can roughly be thought of as the size of the neighborhood of i in which $p_{j|i}$ is high. In our experiments, we used a perplexity of 10 as this proved effective at pulling out small clusters.

One caveat is that the aforementioned transformation uses distances $d_{s_i s_j}$, but the Continuous Jaccard metric outputs a similarity. We map our similarities to distances using the formula:

$$y = \log\left(\frac{1}{0.5\max(x, 0)} - 1\right) \tag{5}$$

See Fig 5 for an illustration of the function. We found that this transformation worked better than naively using $y = 1 - x$, because when the similarity approaches 0, the distance tends to infinity (in practice, the lowest similarity between two seqlets tends to be around 0 rather than around -1).

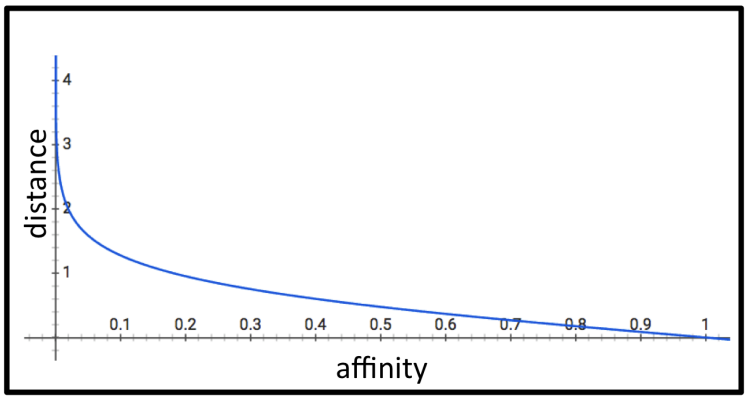

Figure 5: **Plot of $y = \log\left(\frac{1}{0.5\max(x,0)} - 1\right)$, used to map affinities to distances.**

Note that at no point do we compute a lower-dimensional t-sne embedding for clustering purposes. We work directly with the similarities computed in the higher-dimensional space and convert them into a conditional

probability matrix. We then symmetrize the conditional probability matrix by computing the joint probabilities p_{ij} as follows:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N} \tag{6}$$

Where N is the total number of seqlets. This formula for the joint probability of picking both s_i and s_j assumes that the first seqlet s_1 is picked at random and the second seqlet s_2 is picked according to $p_{s_2|s_1}$.

Transformation 2: Averaging Over Several Rounds of Louvain

A desirable choice of community detection algorithm is Louvain community detection [2]. Louvain community detection scales well, does not require the user to pre-specify a number of clusters, and has been used successfully in other biological applications such as PhenoGraph [5]. However, Louvain community detection is not deterministic. We found that if Louvain community detection was applied directly to the joint probabilities described above, the results were somewhat unstable. To minimize this this instability, we follow the results of the first transformation by a second transformation that leveraged the results of repeated rounds of Louvain community detection. Specifically, we run Louvain community detection `louvain_membership_average_n_runs` times (default of 200 times) and defined the affinity between two seqlets to as the fraction of times Louvain community detection placed the two seqlets in the same cluster at the highest level of the hierarchy. We use the implementation of Louvain community detection present in the PhenoGraph package [5].

Clustering the Transformed Matrices

Finally, we subject the affinity matrix produced by the second transformation to Louvain community detection and use the communities produced at the highest level of the hierarchy as our clusters. We found that the second transformation substantially improved the stability of the results compared to using only the first transformation. See the right panel of Fig. 4 for an example of a tsne-embedding visualizing the final clusters picked out by Louvain community detection. To add an additional layer of robustness to random seed, we try different random seeds until we see no improvement in modularity over `contin_runs` consecutive runs of Louvain (default is 50 consecutive runs), and then take the clustering that produced the best modularity.

4.3 Seqlet Aggregation

We aggregate the seqlets within each cluster to generate 'motifs'. To do this, we sort seqlets in descending order of the total magnitude of their importance on all the relevant tasks and merge them together successively in a greedy fashion. Specifically, when merging a seqlet into the aggregated motif, we align the seqlet to all positions where at least `min_overlap_while_sliding` of the seqlet overlaps the motif, and pick the alignment that produces the best Continuous Jaccard similarity. The aggregated motif is then updated by averaging the values of each data track at each position over all the seqlets aligning to that position. This is done until there are no more seqlets left for merging.

4.4 Motif boundary editing

The boundaries of the motifs are then editing in three steps, as described below and illustrated in step (iv) of **Fig. 1**.

(a) Trim away flanks with weak support

Once all the seqlets have been collapsed into a single motif, the boundaries of the motif are trimmed to retain only those positions that have a good number of seqlets covering them. Specifically, at each position in the motif we count the number of seqlets for which the center of the seqlet is aligned to that position. We calculate the number of seqlet centers at the position with the most seqlet centers, and define our threshold to be `frac_support_to_trim_to` (default 0.2) of this value. If this threshold is below `min_num_to_trim_to` (default 30), we set the threshold to `min_num_to_trim_to` instead. Starting from the ends of the motif, we

discard seqlets whose center aligns to positions where the total number of seqlet centers does not not meet the threshold. We stop trimming from a particular end when we encounter a position that passes the threshold.

(b) Expand seqlets to fill their motifs

Once the motif has been trimmed down to retain only those seqlets aligning to positions with good coverage, the ends of the remaining seqlets are expanded on either side to fill the boundaries of the motif as illustrated in step (iv.b) of **Fig. 1**. Note that this expansion step improves the quality of the aggregated motif, because each position in the motif now has information from all the seqlets aligned to the motif.

(c) Center motifs and standardize lengths

For subsequent steps, it is desirable to have motifs that are all of uniform size, even if it means including some uninformative positions in the motif (these uninformative positions can be trimmed away later). To make the motifs have uniform size, each motif is then trimmed to the `trim_to_window_size` bp window (default 30) that has the highest total magnitude of both real and hypothetical importance for all the relevant tasks, and the seqlets in the `trim_to_window_size` bp window are then expanded by `initial_flank_to_add` bp on either size, producing motifs of length `trim_to_window_size` $+ 2 \times$ `initial_flank_to_add`bp.

4.5 Discard motifs that disagree with metacluster activity pattern

After motif boundary editing, the aggregated motif's contribution score pattern may disagree with that of the activity pattern of the metacluster. We have observed that this can be because the original seqlet center was at the noisy flank of a motif where the importance scores on the flank had a different sign than the importance score of on the motif. We filter these motifs out.

4.6 Repeat Seqlet Clustering

In the previous step, the seqlet boundaries are edited and expanded. Often, this can result in each seqlet incorporating additional informative positions that could improve the clustering. Thus, we take the seqlets from all the processed motifs and subject them to a second round of clustering as illustrated in Fig. 1. We found that in practice, this second round of clustering improved the quality and stability of results. In principle, more rounds can be performed, but the current default is to do only one additional round.

5 TF-MoDISco Phase 3: Post-processing of clusters

5.1 Identifying spurious merging

We do a final check to make sure that our Louvain community detection process did not merge two motifs together that one could argue belong in different clusters. To do this, we alter the implementation of Louvain community detection such that, during initialization, each point is randomly assigned to one of two clusters (by contrast, the original initialization assigns each point to its own clusters); this guarantees that at the end of community detection, at most two clusters are returned. We then run Louvain with different random seeds until no improvement in modularity is observed over 20 consecutive random seeds; the clustering that had the best modularity is returned. We refer to this process as "Louvain diclustering".

The spurious merging detection proceeds recursively as follows:

- For each motif, we first check to see whether the motif contains more than `final_min_cluster_size` (default 30) seqlets; if it does, we inspect the motif for spurious merging. we look at the constituent seqlets and compute the similarity between seqlets using the Continuous Jaccard Similarity metric (**Eqn. 3**). This is similar to the process used to compute the original fine-grained affinity matrix, except that we look at all seqlet-seqlet pairs and we do not need to look at different alignments of the seqlets, as we rely on the seqlet alignment that was used to aggregate the seqlet into the motif.
- Given the seqlet-seqlet similarity matrix for seqlets in a motif, we run Louvain diclustering. If the diclustering produces more than one subcluster, we aggregate the seqlets within the subclusters to produce sub-motif.

- We check whether the sub-motifs are substantially dissimilar by looking at their Pearson correlation. If the Pearson correlation is less than `threshold_for_spurious_merge_detection` (default 0.8), we declare the two sub-motifs to be dissimilar and repeat the spurious merging detection on each sub-motif. Otherwise, we terminate the recursion.

5.2 Merging Redundant patterns

Having performed spurious merging detection, we now collapse similar motif clusters together to obtain a final list of non-redundant motifs. We will use two points of information when deciding whether to merge: one is the maximum cross-correlation (after mean and magnitude normalizing the score tracks) over all alignments of the motifs that overlap by at least `min_overlap_while_sliding` (default 0.7), and the other is a measure of similarity that takes into account how tightly packed the motif is. We find that cross-correlation performs better than cross Continuous Jaccard at this stage because the motifs are aggregated over several seqlets and thus the scores are less noisy. We describe the second measure in more detail in the next section.

Density-sensitive similarity

In Sec. 4.2, we used conditional probabilities inspired by t-sne to adapt our notion of distance to the local density of the data. We wish to leverage a similar idea when computing the similarity between two motif clusters. Let m_i denote a motif. For all seqlets s_j that were clustered to form the list of motifs, we define $p_{j|i}$ as:

$$p_{j|i} \propto \exp\big(-\beta_i d_{s_j m_i}\big) \tag{7}$$

Note that this formula is analogous to eqn. 4. Here, $d_{s_j m_i}$ is the distance from seqlet s_j to motif m_i, computed using the Continuous Jaccard similarity subject to the transformation in eqn. 5. The purpose of this is to calculate the value of β_i such that the perplexity of $p_{s|m_i}$ is equal to the number of seqlets that aligned to motif m. We then define the density-sensitive similarity as:

$$\text{DensitySensitiveSim}_{m_i m_j} = \exp\big(-\max(\beta_i, \beta_j) d_{m_j m_i}\big) \tag{8}$$

As high values of β indicate tighter packing, this can be thought of using the more tightly-packed motif cluster to calculate the density-adjusted similarity to the other motif.

Iterative motif merging

Once we have our motif similarities from cross-correlation as well as our density-sensitive similarities, we are ready to merge motifs together. We wish to collapse highly similar motifs together, but our idea of similarity may not be transitive (that is, if m_1 is similar to m_2 and m_2 is similar to m_3, it is not necessary that m_1 is sufficiently similar to m_3 that we would be comfortable merging m_1, m_2 and m_3 together). To avoid merging dissimilar motifs together, we will perform our motif merging use a combination of two criteria: one criterion for whether two motifs are similar enough to attempt to merge them, and another for whether two motifs are sufficiently dissimilar that we do not want them merged together.

We use an iterative algorithm as follows: Let S_i denote the set of motifs that are to be merged with motif i, including i itself; at the beginning of an iteration, this is just a set containing the single element i. We enumerate all possible motif pairs (i, j) and iterate over the pairs in descending order of their cross-correlation similarity. If the combination of the cross-correlation similarity and the density-sensitive similarity meet our predefined criterion potentially merging i and j, we proceed to iterate over all the motifs currently in S_i and S_j and make sure that no pair of motifs meet our criterion for incompatibility. If no incompatibilities our found, we update our sets to reflect the fact that all the motifs in S_i and S_j are going to be merged together. At the end of the iteration, we aggregate all the seqlets in the motifs of each set and repeat the process until we reach an iteration where no motifs are merged together.

The criteria for motifs being considered similar or dissimilar can be modified by the user. Let p represent the density-sensitive similarity between two motifs, and let c represent the average cross-correlation-based similarity. The default criterion for motifs to be considered similar is that at least one of the following must

be true: ($p > 0.0001$ and $c > 0.84$), or ($p > 0.00001$ and $c > 0.87$), or ($p > 0.000001$ and $c > 0.90$). The default criterion for motifs to be considered dissimilar is that at least one of the following must be true: ($p < 0.1$ and $c < 0.75$), or ($p < 0.01$ and $c > 0.8$), or ($p < 0.001$ and $c < 0.83$), or ($p < 0.0000001$ and $c < 0.9$). These defaults were set based on empirical observations. Future versions of TF-MoDISco will likely refine this motif merging step.

5.3 Reassign seqlets from small clusters to the dominant clusters

We disband motifs for which the number of seqlets in the motif falls below `final_min_cluster_size` (default 30). The seqlets in the disbanded motifs are assigned to whichever of the remaining motifs they have the highest affinity to (as measured by Continuous Jaccard similarity), provided that the similarity exceeds `min_similarity_for_seqlet_assignment` (default 0.2).

5.4 Final flank expansion

The motifs are ultimately expanded to reveal potential flanking patterns. In this work, we expand the motifs to include an additional `final_flank_to_add` (default 10) bp on either side, for a total of `trim_to_window_size` $+ 2 \times$ (`initial_flank_to_add` + `final_flank_to_add`)bp. Note that flanks can be trimmed as desired if they are deemed to be uninformative.

Author Contributions

Avanti Shrikumar conceived of and was the primarily developer of TF-MoDISco. Katherine Tian, Anna Shcherbina, Žiga Avsec and Surag Nair contributed features to the TF-MoDISco codebase. Anna Shcherbina proposed the idea of fast affinity matrix computation using kmers. Abhimanyu Banerjee, Mahfuza Sharmin and Žiga Avsec applied versions of TF-MoDISco immediately leading up to v0.4.4.2-alpha on biological data and identified areas of improvement. Anshul Kundaje provided guidance and feedback. Avanti Shrikumar wrote this technical note.

Acknowledgments

We thank Stefan Holderback for his feature suggestion and pull request to v0.4.4.2-alpha of TF-MoDISco. We thank Nasa Sinnott-Armstrong for writing an initial version of GPU-based cross-correlation which was used in early prototypes of TF-MoDISco. We thank Chuan Sheng Foo for exploring k-means based clustering approaches for early prototypes of TF-MoDISco. We thank Chuan Sheng Foo, Johnny Israeli, Peyton Greenside and Irene Kaplow for evaluating early prototypes of TF-MoDISco on biological data.

References

[1] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of dna-and rna-binding proteins by deep learning. *Nature biotechnology*, 2015.

[2] Vincent D Blondel, Jean-Lou Guillaume, Renaud Lambiotte, and Étienne Lefebvre. The louvain method for community detection in large networks. *J. Stat. Mech: Theory Exp.*, 10:P10008, 2011.

[3] David R Kelley, Jasper Snoek, and John L Rinn. Basset: Learning the regulatory code of the accessible genome with deep convolutional neural networks. *Genome research*, 2016.

[4] Jack Lanchantin, Ritambhara Singh, Zeming Lin, and Yanjun Qi. Deep motif: Visualizing genomic sequence classifications. *arXiv preprint arXiv:1605.01133*, 2016.

[5] Jacob H Levine, Erin F Simonds, Sean C Bendall, Kara L Davis, El-Ad D Amir, Michelle D Tadmor, Oren Litvin, Harris G Fienberg, Astraea Jager, Eli R Zunder, Rachel Finck, Amanda L Gedman, Ina Radtke, James R Downing, Dana Pe'er, and Garry P Nolan. Data-Driven phenotypic dissection of AML reveals progenitor-like cells that correlate with prognosis. *Cell*, 162(1):184–197, July 2015.

[6] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *J. Mach. Learn. Res.*, 9(Nov):2579–2605, 2008.

[7] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. 70:3145–3153, 06–11 Aug 2017.

[8] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *CoRR*, abs/1703.01365, 2017.