Structural Motifs Documentation

Release 0.1

Peter DeFord

Jan 30, 2018

CONTENTS

Structural Motifs provide a way to represent DNA motifs based on the shape of the DNA instead of the DNA sequence. These motifs may represent the preferences driving specificity of transcription factors for their binding sites. For more information about the underlying model of StruMs, refer to *Introduction to Structural Motifs*.

The package described here provides a framework for computing and working with StruMs, based on DNA shape features from the Dinucleotide Property DataBase. Currently available capabilities include:

- Train models from known binding sites
- Use expectation maximization to find *de novo* motifs
- Provide utility to incorporate additional quantitative features, e.g. DNase hypersensitivity.
- Score matches of the motif to novel sequences

CHAPTER

ONE

CONTENTS

1.1 Introduction to Structural Motifs

It has long been observed that transcription factors show a higher affinity to some sequences than others, and seem to tolerate some variability around these highest-affinity sequences.

The interaction between sequence specific transcription factors and DNA requires compatability at a complex interface of both electrostatics and 3D structure. Analagous to sequence representations of transcription factor binding sites, we assume that transcription factors have a preference for a specific shape of the DNA across the binding site and will tolerate some variation around that preferred shape.

Representing the values that a given position-specific shape feature may adopt as v_j, we expect the preference of the transcription factor to be normally distributed:

$$v_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$$

In which case, given a set D of n sequences with m position-specific features that describe a sample of the binding sites preferred by a given transcription factor:

$$D = \begin{Bmatrix} (v_{11}, & v_{12}, & ... & v_{1m}), \\ (v_{21}, & v_{22}, & ... & v_{2m}), \\ ... & ... & ... & ... \\ (v_{n1}, & v_{n2}, & ... & v_{nm}), \end{Bmatrix}$$

we can compute a set of parameters ϕ describing the specificity of that transcription factor:

$$\phi = \begin{Bmatrix} (\mu_1, \sigma_1), \\ (\mu_2, \sigma_2), \\ ... \\ (\mu_m, \sigma_m) \end{Bmatrix}$$

If we also assume that each feature and each position is independent, then calculating the score s for the *i*th sequence becomes:

$$s_i = \prod_{j=1}^{m} P(v_{ij}|\mu_j, \sigma_j^2)$$

In order to avoid underflow issues during computation, all computations are done in log space.

1.1.1 *De novo* Motif finding

For *de novo* motif finding, an expectation maximization approach is employed. This approach assumes that there is exactly one occurrence of the binding site on each of the training sequences. This based on the OOPS model employed by MEME.

E-step

The likelihood (l_{ij}) of the jth position in the ith sequence being the start of the binding site is taken to be the score of the StruM at that position multiplied by the likelihood of the flanking regions matching the background model (ϕ_B):

$$l_{ij} = \prod_{n=1}^{j-1} P(v_{ij}|\phi_B) \prod_{n=j}^{j+k-1} P(v_{ij}|\phi_{i-j+1}) \prod_{n=j+k}^{N} P(v_{ij}|\phi_B)$$

The likelihoods are then normalized on a by-sequence basis to produce M, the matrix of expected start positions:

$$M_{ij} = \frac{l_{ij}}{\sum_{j'=1}^{m} l_{ij'}}$$

M-step

The maximization step takes these likelihoods and calculates maximum likelihood values for μ and σ for each of the m position-specific features:

$$\mu_j = \sum_{i=1}^{n} \sum_{\mathrm{v}} \frac{v_{ij} \cdot M_{ij}}{\sum_i \sum_j M_{ij}}$$

$$\sigma_j = \sum_{i=1}^{n} \sum_{\mathrm{v}} \frac{(v_{ij} - \mu_j)^2 \cdot M_{ij}}{\sum_i \sum_j M_{ij} - \frac{\sum_i \sum_j M_{ij}^2}{\sum_i \sum_j M_{ij}}}$$

1.2 Installation Instructions

1.2.1 Manual Installation of Dependencies

The StruM package depends on the following libraries:

- numpy
- pandas
- scipy
- matplotlib
- bx-python

1.2.2 Installing dependencies with Conda

By far the easiest way to install all of the dependencies is with conda. With conda installed, the dependencies can be installed with:

```
conda install numpy scipy pandas matplotlib
conda install -c bioconda bx-python
```

1.2.3 Installing dependencies with PyPI

Another method for installing the dependencies manually is using the pip command from *PyPI <https://pypi.python.org/pypi>*. Then the dependencies can be installed with:

```
pip install numpy scipy pandas matplotlib bx-python
```

1.2.4 Installation of StruM Package

To install the StruM package, simply download the source code, navigate to that directory, and run:

```
python setup.py install
```

1.3 Examples

This page contains simple examples showcasing the basic functionality of the StruM package.

Contents

1.3.1 Basic Example

The basic usage of the StruM package covers the case where the binding site is already known. If the PWM is known already, a tool such as FIMO could be used to determine possible matches to the binding site.

In this case, a maximum likelihood StruM can be computed directly from these sequences. Source code for this example can be found here: *basic.py*.

```
#!/usr/bin/env python

# Imports
import numpy as np
import strum

# Sequences representing some of the variability of the
# FOXA1 binding site.
training_sequences = [
    "CTGTGCAAACA",
    "CTAAGCAAACA",
    "CTGTGCAAACA",
```

```
    "CTGTGCAAACA",
    "CAGAGCAAACA",
    "CTAAGCAAACA",
    "CTGTGCAAACA",
    "CAATGTAAACA",
    "CTGAGTAAATA",
]

# Initialize a new StruM object, using the basic features
# from the DiProDB table.
motif = strum.FastStruM(mode='basic')

# Use the training sequences to define the StruM,
# ensuring that that the variation of all position-specific
# features is at least 10e-5 (lim)
motif.train(training_sequences, lim=10**-5)

# Print out what the PWM would look like for these sequences
motif.print_PWM(True)

# Define an example sequence to analyze.
test_sequence = "ACGTACTGCAGAGCAAACAACTGATCGGATC"
# Reverse complement it, as the best match may not be on
# the forward strand.
reversed_test = motif.rev_comp(test_sequence)

# Get a score of the similarity for each kmer in the test
# sequence to the StruM.
forward_scores = motif.score_seq(test_sequence)
reverse_scores = motif.score_seq(reversed_test)

# Find the best match.
## Determine the strand the best match lies on
if np.max(reverse_scores) > np.max(forward_scores):
    score, seq = reverse_scores, reversed_test
    strand = "-"
else:
    score, seq = forward_scores, test_sequence
    strand = "+"
## Find position in the sequence
best_pos = np.argmax(score)
best_seq = seq[best_pos : best_pos + motif.k]
best_score = score[best_pos]

# Print the results to screen
print "\n", strand, best_pos, best_seq, best_score
```

You should see the position weight matrix that would be derived from these sequences followed by the strand, position, matching kmer, and score for the best match in the test sequence to the StruM. The labels surrounding the PWM can be toggled off by changing the `labels` parameter in *strum.FastStruM.print_PWM()*.

```
    1     2     3     4     5     6     7     8     9    10    11
A 0.000 0.222 0.333 0.444 0.000 0.000 1.000 1.000 1.000 0.000 1.000
C 1.000 0.000 0.000 0.000 0.000 0.778 0.000 0.000 0.000 0.889 0.000
G 0.000 0.000 0.667 0.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000
T 0.000 0.778 0.000 0.556 0.000 0.222 0.000 0.000 0.000 0.111 0.000

+ 8 CAGAGCAAACA -43.2745526744
```

1.3.2 *De Novo* Motif Finding Example

More often, the task is to take a set of sequences and perform *de novo* motif finding, simultaneously modeling the motif and identifying the putative binding sites. The StruM package allows you to do this using expectation maximization.

The following example uses the EM capabilities of the StruM package to identify the motif GATTACA randomly inserted into 50 sequences. The source gode for this example can be downloaded here: *em.py*.

Refer to `strum.FastStruM.train_EM()` for more info on the parameters for the EM algorithm.

```
#!/usr/bin/env python

# Imports
import numpy as np
import strum

# 50 example sequences that contain the sequence GATTACA
training_sequences = [
        "CGATTACAGATCTCCCGCGACCCTT", "GATGATTACAAGATGCGTCGAATAT",
        "AGCCCTGTCCGCAGATTACAACCAC", "AGTTAACTCCCTAGATTACATTTGT",
        "CGCTACAAAGTAAAGGAGATTACAT", "CTGATTACATCCTGTCCGAGGCGTG",
        "CGATTACATTCAACTAGATGCGCGC", "GAACGGCATGGGCGATTACAACACT",
        "CGGGGTGATCGTAATGTGATTACAA", "TGGTGCCCTGATTACACCTTACATG",
        "GGCGAAGATTACATCCTCGGCCCAT", "ATCGGATTACATGTACTCGTCCACG",
        "TCGGGGATTACAGGGGAGACGCTTA", "AGGTAGATTACATCGTTTTATTAGT",
        "TATTGTGCTCGATTACAAGCAGGCC", "CAGACCGCTTACACGTTGATTACAA",
        "GACACCCTCGATTACACCTCGTATA", "GGAACCGCGCGGATTACACGCGAGA",
        "GTTGATTACAAGGGAAACATACTTG", "CCCACACATTAGCTCGAGATTACAT",
        "GCAGAGTACCCTGCGGCGATTACAA", "ATACTCACGCATACAGATTACAAGA",
        "GGTATGCATCGCGATTACAGCACTG", "GGATTACAGTGAGCCTGCACCTTGA",
        "TTGGATTACATGGCCAAACTCCACT", "GGCCGGCAGAGATTACACTAGAGAG",
        "ACAGATTACAGTGCAAATTGAGCAG", "CCACTGCATGACTGGATTACAGGCA",
        "CATGCCGGCGGTTAACGGATTACAC", "CCGATTACAAGTGCTCTGCACGGCG",
        "CATATAGAGGCGATTACAGCGTATC", "GGAACGATTACAGTGAGACTGCTCC",
        "ATGATTACAGCGAAACGTATTCAAA", "TTTTCGGATGATTACACATTCTTCT",
        "GTACAATGCATCGCGATTACAACAC", "GGATTACAAGTATCTGCCTGGATAC",
        "CTCCCGATTACATCAGGTACGTCCT", "TAGAGAAGATTACAGCCTACTATTG",
        "AAGCTTTGGGCCGTACGATTACATC", "GTAAGATTACAAGTTCAGGGTGATC",
        "CATGATTACATTGGCGCCGACCTAC", "GCTGGATTACAATCATACCCGTGTA",
        "GGTTAGGGATTACAAACAAGACGTG", "GACCGAGGTCTGATTACACTCCATC",
        "ATAGACGCGATTACAAGCACTCTAA", "TTTCCGTTCTGCAGCTGATTACAAC",
        "GGATTACACGCCTTCTCAAGCAGTG", "ATCCTAACAGGATTACAAGAATTAC",
        "TATGAAGCTGAAGAAGATTACAGCA", "CCTGTCTCAGATTACAGCACGGCGG",
]

# Initialize a new StruM object, using the DNA groove related
# features from the DiProDB table. Specify to use 4 cpus
# when doing EM.
motif = strum.FastStruM(mode='groove', n_process=4)

# Train the model on the training sequences using expectation
# maximization, ensuring that that the variation of all
# position-specific features is at least 10e-5 (lim). Use
# a random_seed for reproducibility, and repeat with 20
# random restarts.
k = 8
```

```
motif.train_EM(training_sequences, fasta=False, k=k,
    lim=10**-5, random_seed=620, n_init=20)

# Examine the output, by identifying the best matching
# kmer in each of the training sequences.
out = []

for sequence in training_sequences:
    rseq = motif.rev_comp(sequence)
    s1 = motif.score_seq(sequence)
    s2 = motif.score_seq(rseq)
    i1 = np.argmax(s1)
    i2 = np.argmax(s2)
    if s1[i1] > s2[i2]:
        seq = sequence
        i = i1
        s = "+"
    else:
        seq = rseq
        i = i2
        s = "-"
    out.append(seq[i:i+k])
        print "{}{: <2} {: >{}} {} {}".format(
                s, i, seq[:i].lower(), len(seq)-k,
                seq[i:i+k].upper(), seq[i+k:].lower()
                )

# Summarize these best matches with a simple PWM.
nucs = dict(zip("ACGT", range(4)))
PWM = np.zeros([4,k])
for thing in out:
    for i,n in enumerate(thing):
        PWM[nucs[n], i] += 1

PWM /= np.sum(PWM, axis=0)
for row in PWM:
    print row
```

There are several sections that are reported in the output. First, the StruM gives a live update on how many iterations it took to converge for each of the random restarts. Once they have all completed, it shows the likelihoods for each of the restarts. The example output below highlights why it is important to repeat this process a number of times, as the results are highly variable. Finally we output a summary of the results. It is obvious that the model correctly identified the sequence NGATTACA as observed both in the highlighted sequences and the summary PWM at the bottom.

```
Retaining 50 out of 50 sequences, based on length (>25bp)
Detected cyclical likelihoods. Proceeding to max.
Converged after 10 iterations on likelihood
Converged after 5 iterations on likelihood
...
Restart Likelihoods: [1981.3168691613439, 1969.7041022864935, 1969.7041022864935, 650.
↪89659719217332, 386.24238189057303, 369.62562708240256, 350.54071957020324, 350.
↪54071957020324, 350.54071957020324, 327.8331936774872, 315.95426601483587, 242.
↪73205512692093, 235.90375348128853, 149.92555284207441, -15.089679773020009, -68.
↪060441249738645, -164.9202144304158, -373.49851273275601, -524.90598178843163, -524.
↪91730067288267]
+0                    CGATTACA gatctcccgcgaccctt
+2                  ga TGATTACA agatgcgtcgaatat
+12      agccctgtccgc AGATTACA accac
```

```
+12      agttaactccct AGATTACA tttgt
...
[ 0.26  0.    1.    0.    0.    1.    0.    1.  ]
[ 0.28  0.    0.    0.    0.    0.    1.    0.  ]
[ 0.24  1.    0.    0.    0.    0.    0.    0.  ]
[ 0.22  0.    0.    1.    1.    0.    0.    0.  ]
```

1.3.3 Adding Additional Features

With the StruM package, in additional to the structural features provided in the DiProDB table, you can incorporate additional arbitrary features, as long as they provide a quantitative value across the binding site.

In the example below we will define a function that looks up a DNase signal from a `bigwig` file to incorporate. The file we will be using comes from a DNase experiment in K562 cells, mapped to *hg19* from the ENCODE project (ENCFF111KJD) and can be downloaded from here.

Warning: Relies on the older version of the StruM (for now) Deprecated with FastStruM (for now)

The source code for this example can be found here: *DNase.py*.

```
# Imports
import numpy as np
import sys

import bx.bbi.bigwig_file
import StruM

# Specify the path to where you downloaded the bigwig
# file. E.g. "/Users/user/Downloads/ENCFF111KJD.bigWig"
DNase_bigwig_path = sys.argv[1]

# Define the function to be used by the StruM when
# converting from sequence-space to structural-space.
# NOTE: This function takes additional parameters.
def lookup_DNase(data, chrom, start, end):
        """Lookup the signal in a bigWig file, convert NaNs to
        0s, ensure strandedness, and return the modified signal.

        Parameters:
                data : (str) - Path to the bigWig file.
                chrom : (str) - Chromosome name.
                start : (int) - Base pair position on the chromsome of
                                                the beginning of the region of␣
↪interest.
                end : (int) - Base pair position on the chromsome of
                                        the end of the region of interest.

        Returns:
                trace : (1D array) - The signal extracted from the
                                                        bigWig file for the region of
                                                        interest.
        """
        # Open file
        bwh = bx.bbi.bigwig_file.BigWigFile(open(data))
        # Lookup signal for regions
```

```
        trace = bwh.get_as_array(chrom, min(start,end), max(start, end)-1)
        # Clean up NaNs
        trace[np.isnan(trace)] = 0.0
        # Ensure strandedness
        if start > end:
                trace = trace[::-1]
        return trace

# Some example sequences and their chromosome positions,
# from human build hg19.
training_data = [
        ['GAGATCCTGGGTTCGAATCCCAGC', ('chr6', 26533165, 26533141)],
        ['GAGATCCTGGGTTCGAATCCCAGC', ('chr19', 33667901, 33667925)],
        ['GAGGTCCCGGGTTCGATCCCCAGC', ('chr6', 28626033, 28626009)],
        ['GAGGTCCTGGGTTCGATCCCCAGT', ('chr6', 28763754, 28763730)],
        ['GGGGGCGTGGGTTCGAATCCCACC', ('chr16', 22308468, 22308492)],
        ['GGGGGCGTGGGTTCGAATCCCACC', ('chr5', 180614647, 180614671)],
        ['AAGGTCCTGGGTTCGAGCCCCAGT', ('chr11', 59318028, 59318004)],
        ['GAGGTCCCGGGTTCAAATCCCGGA', ('chr1', 167684001, 167684025)],
        ['GAGGTCCCGGGTTCAAATCCCGGA', ('chr7', 128423440, 128423464)],
]

# Initialize a new StruM object.
motif = StruM.StruM()

# Update the StruM to incorporate the function
# defined above, drawing on the bigWig as the
# data source.
motif.update(data=DNase_bigwig_path, func=lookup_DNase,
        features=['k562_DNase'])

# Train the model using the modified StruM and the example
# data above
motif.train(training_data)

# Evaluate the similarity to the model of a new sequence.
seq = 'GAGGTCCCGGGTTCAATCCCCGGC'
position = ['chr2', 157257305, 157257329]

rseq = motif.rev_comp(seq)
rposition = [position[0], position[2], position[1]]

s = [
        motif.score_seq(seq, *position),
        motif.score_seq(rseq, *rposition)
]

strands = ['-', '+']
print "Best match found on the '{}' strand".format(strands[np.argmax(s)])
```

1.4 Documentation

1.4.1 StruM: Structural Motifs

This package provides functionality for computing structural representations of DNA sequence motifs. Estimates for DNA structure comes from the DiNucleotide Property Database (http://diprodb.leibniz-fli.de/).

class strum.**FastStruM**(*mode='full'*, *n_process=1*, *custom_filter=[]*, *func=None*)

Differs from the standard StruM in that it cannot incorporate additional features, it requires all sequences to have the same length (or else will filter to the shortest length), and uses heuristics to speed up scoring.

__init__(*mode='full'*, *n_process=1*, *custom_filter=[]*, *func=None*)

Create a FastStruM object.

Parameters

- **mode** (*str.*) – Defines which subset of available features in the DiProDB table to use. Choose from: ['basic', 'groove', 'protein', 'full', 'nucs', 'unique', 'proteingroove', 'custom']
- **custom_filter** (*list of ints.*) – Specifies the indices of desired features from the DiProDB table.
- **n_process** (*int.*) – Number of threads to use. `-1` uses all processers.
- **func** (*function.*) – Additional scoring functions to incorporate.

calc_z(*x*, *mu*, *var*)

Calculate Z-scores for values based on mean and standard deviation.

Parameters

- **x** (*float, numpy array.*) – Value or values of interest.
- **mu** (float, numpy array (`x.shape[1]`).) – Average of population from which `x` was sampled.
- **var** (float, numpy array (`x.shape[1]`).) – Variance of population from which `x` was sampled.

Returns The Z-score for all values in `x`.

Return type `type(x)`

define_PWM(*seqs*, *weights=None*)

Computes a position weight matrix from sequences used to train the StruM.

Parameters

- **seqs** (*list of str.*) – Training set, composed of gapless alignment of binding sites of equal length.
- **weights** (*1D array of floats.*) – Weights to associate with each of the sequences in `seqs` to use in learning the motif.

Returns None. Sets the position weight matrix `self.PWM` based on the weighted sequences.

eval(*struc_kmer*)

Compares the structural representation of a sequence to the StruM.

Parameters **struc_kmer** (output of *translate()*.) – A kmer that has been translated to structure-space via *translate()*.

Returns *log* score of similarity of kmer to StruM.

Return type float.

norm_p(*x*, *mu*, *var*)
Finds the one-tail p-value for values from the standard normal distribution. Adds a 'pseudocount' of 10e-300 to avoid underflow.

Parameters

- **x** (*float, numpy array.*) – Value or values of interest.
- **mu** (float, numpy array (`x.shape[1]`).) – Average of population from which `x` was sampled.
- **var** (float, numpy array (`x.shape[1]`).) – Variance of population from which `x` was sampled.

Returns The p-value for all values in `x` relative to `mu` and `var`.

Return type `type(x)`

print_PWM(*labels=False*)
Pretty prints the PWM to std_out.

Parameters **labels** (*bool.*) – Flag indicating whether to print the PWM with labels indicating the position associated with each column, and the nucleotide associated with each row.

Returns Formatted position weight matrix suitable for display, or use in the MEME suite, e.g. Also prints the PWM to `std_out`.

Return type str.

read_FASTA(*fasta_file*)
Reads a FASTA formatted file for headers and sequences.

Parameters **fasta_file** – FASTA formatted file containing DNA sequences.

Type file object

Returns The headers and sequences from the FASTA file, as two separate lists.

Return type (list, list)

rev_comp(*seq*)
Reverse complement (uppercase) DNA sequence.

Parameters **seq** (*str.*) – DNA sequence, all uppercase characters, composed of letters from set ACGTN.

Returns Reverse complement of `seq`.

Return type str.

score_seq(*seq*)
Scores a sequence using pre-calculated StruM.

Parameters **seq** (*str.*) – DNA sequence, all uppercase characters, composed of letters from set ACGTN.

Returns Vector of scores for similarity of each kmer in `seq` to the StruM.

Return type 1D array.

train(*training_sequences*, *weights=None*, *lim=None*)
Learn structural motif from a set of known binding sites.

Parameters

- **training_sequences** (*list of str.*) – Training set, composed of gapless alignment of binding sites of equal length.
- **weights** (*1D array of floats.*) – Weights to associate with each of the sequences in `training_sequences` to use in learning the motif.
- **lim** (*float*) – Minimum value allowed for variation in a given position-specific-feature. Useful to prevent *any* deviation at that position from resulting in a probability of 0.

Returns None. Defines the structural motif `self.strum` and the corresponding position weight matrix `self.PWM`.

train_EM(*data*, *fasta=True*, *params=None*, *k=10*, *max_iter=1000*, *convergence_criterion=0.001*, *random_seed=0*, *n_init=1*, *lim=None*, *seqlength=None*)

Performs Expectation-Maximization on a set of sequences to find motif.

Parameters

- **data** (*list of str, open file object referring to a FASTA file.*) – A set of sequences to use for training the model. Assumed to have one occurrence of the binding site per sequence.
- **fasta** (*bool.*) – Flag indicating whether `data` points to an open file object containing a FASTA formatted file with DNA sequences.
- **params** (`*args`, `**kwargs`.) – Additional parameters to pass to `self.func`, if defined.
- **k** (*int.*) – Size of binding site to consider. Since dinucleotides are considered, in sequence-space the size of the binding site will be `k + 1`.
- **max_iter** (*int.*) – Maximum number of iterations of Expecation Maximization to perform if convergence is not attained.
- **convergence_criterion** (*float.*) – If the change in the likelihood between two iterations is less than this value, the model is considered to have converged.
- **random_seed** (*int.*) – Seed for the random number generator used in the EM algorithm for initialization.
- **n_init** (*int.*) – Number of random restarts of the EM algorithm to perform.
- **lim** (*float*) – Minimum value allowed for variation in a given position-specific-feature. Useful to prevent *any* deviation at that position from resulting in a probability of 0.
- **seqlength** (*int.*) – If set, the sequences in the training data will be trimmed symmetrically to this length. .. note:: This must be longer than the shortes sequence.

Returns None. Defines the structural motif `self.strum` and the corresponding position weight matrix `self.PWM`.

translate(*seq*)

Convert sequence from string to structural representation.

Parameters **seq** (*str.*) – DNA sequence, all uppercase characters, composed of letters from set ACGTN.

Returns Sequence in structural representation.

Return type 1D numpy array of floats.

`strum.`**read_diprodb**()

Load the values from the DiNucleotide Property Database as a lookup table.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

C

D

E

F

N

P

R

S

T