string-kernel Documentation

Release 1.0

Meng Zhang,Mitra Darvish,Jakob Hertzberg

Jan 07, 2020

Contents

CHAPTER 1

Getting Started

1.1 Introduction

Strkernel is a python package designed to perform a kernel based analysis of biological sequences. The implementation assumes the use of Support Vector Machines (SVMs) but does not strictly require it since each kernel can be used separately of any machine learning algorithm. Further instructions on how to combine the kernel methods with machine learning approaches can be found in the Tutorials section. The package provides three different kernels which can be used for sequence analysis:

- Gappy Kernel[1]
- Mismatch Kernel[2]
- Motif Kernel[3]

The general use case of strkernel is the conversion of a set of sequences into a matrix with numeric similarity measurements. The expected input are either sequences as strings or Sequence objects of the Biopython package. In addition to the default output, the Motif Kernel implementation supports the output of a similarity matrix for a set of given sequences. The default output, however, is a sparse matrix where each row represents one sequence. The meaning of the columns depends on which kernel was used and is be explained in the corresponding section.

1.2 Installation

- Install strkernel via PyPi (recommended):

```
sudo pip install strkernel
```

- Install strkernel via github:

[1] Pavel Kuksa, Pai-Hsi Huang, and Vladimir Pavlovic. A fast, large-scale learning method for protein sequence classification. In 8th Int. Workshop on Data Mining in Bioinformatics, pages 29–37, Las Vegas, NV, 2008.

[2] Christina S. Leslie, Eleazar Eskin, Adiel Cohen, Jason Weston, and William Stafford Noble. Mismatch string kernels for discriminative protein classification. Bioinformatics, 1(1):1–10, 2003.

[3] Asa Ben-Hur and Douglas L. Brutlag. Remote homology detection: a motif based approach. Bioinformatics, 19:26–33, 2003.

```
git clone https://github.com/jakob-he/string-kernel
python setup.py install
```

1.3 Tests

You can run several tests to ensure that the package is working. To use the testsuite you have to download the package via github:

```
git clone https://github.com/jakob-he/string-kernel
python setup.py test
```

1.4 What is a String-Kernel?

To use SVMs with strings it is necessary to have a measurement of similarity for different strings. The idea of the most basic string kernel, the spectrum kernel is to count the appearance of k-mers in the string. As a result a string can be represented as a numerical sequence and the similarity of two strings can be calculated by using different kernels, for instance the dot product (linear kernel) of these numerical representations can be calculated. Bio sequences such as DNA, RNA and protein sequences are widely used in SVMs, because they have some certain differences to normal text strings (e.g. a smaller alphabet) kernels fitted to their need were invented. Suchs as the gappy-pair, mismatch and motif kernel.

1.5 References

CHAPTER 2

Kernels

2.1 Mismatch Kernel

The mismatch kernel can be used with support vector machines (SVMs) in a discriminative approach to a classification problem. It measures the sequence similarity based on shared occurrences of **k**-length subsequences, counted with up to **m** mismatches, which is what we generally state as **(k, m)-mismatch**. The kernels can be efficiently computed by using a mismatch tree data structure[1].

The user input consists of the following elements:

- data: 2D matrix of shape (n_samples, n_features), sequences needed to be computed as kernel
- l: int, size of alphabet. Examples of values with a natural interpretation:

  2: for binary data

  256: for data encoded as strings of bytes

  4: for DNA/RNA sequence data (bioinformatics)

  20: for protein data (bioinformatics)
- k: int, used in k-mers to compute the kernel
- m: int, maximum number of mismatches for 2 k-mers to be considered 'similar'. Normally, small values of m should work well. The complexity of the algorithm is exponential in m

The expected output is the mismatch string kernel:

- kernel: 2D matrix of shape (n_samples, n_samples) suggesting the similarities between sequences

2.1.1 How to use the Mismatch Kernel

The collection of computed sequences has to be defined as a list of strings, for example:

[1] Leslie C.S., Eskin E., Cohen A., Weston J., Noble W.S. Mismatch string kernels for discriminative protein classification. Bioinformatics. 2004;20:467–476. doi: 10.1093/bioinformatics/btg431.

```
mismatch_collection = ['aATGCg', 'ACGTTT', 'agATGC','TCACcg', 'cgTCTCGAgt']
```

This collection can then be used to compute the mismatch kernel:

```
from strkernel.mismatch_kernel import MismatchKernel
mismatch_kernel = MismatchKernel(l=l, k=k, m=m).get_kernel(mismatch_collection)
```

The sequence collection has to be preprocessed to regulate the input format and avoid subsequent errors before being used to compute the mismatch kernel, which can be achieved by calling the function *preprocess* in *mismatch_kernel*:

```
from strkernel.mismatch_kernel import preprocess
after_process = preprocess(mismatch_collection)
mismatch_kernel = MismatchKernel(l=l, k=k, m=m).get_kernel(after_process)
```

The result is a sparse matrix indicating the similarities between different sequences based on the inner product of occurence counts of all (k, m)-mismatch k-mers. If the lower case letters in every string are being considered, the parameter *ignoreLower* in *preprocess* function should be set as **False**, which is **True** by default. In addition, the function *get_kernel* can also be tuned by the parameter *normalize*. Normalization is enabled by default since it generally improves accuracy. Normalization is realized by: **kernel[x, y] / sqrt(kernel[x, x] * kernel[y, y])**. That is, the diagonal elements in the kernel matrix are set to 1 since the similarity between a certain string and itself is the largest:

```
sequences = ['aatgcACGTTGAgatcg','acgtgACGTTTGacggt', 'agtccATGCTGTaagtc',
↪'gttccTCACCGTcgcgt', 'gtacgTCTCGCTgtcgt']
# preprocess
after_process = preprocess(mismatch_collection, ignoreLower=False)
# compute mismatch kernel
mismatch_kernel = MismatchKernel(l=l, k=k, m=m).get_kernel(after_process, normalize =␣
↪False)
```

We also provide a function to display the middle step of getting a kernel allowing the users to check the details before computing the final kernel. *leaf_kmers* shows us all mismatch kmers and the occurence counts of every (k, m)-mismatch k-mer in every string. n vectors of length m are the output of *leaf_kmers*, where n is number of strings, whose similarities will be computed in the susequent steps and m is the number of all (k. m)-mismatch k-mers. The similarity between string i and string j is the inner product of vector i and j, where 1<= i, j <= n:

```
print(mismatch_kernel.leaf_kmers)
```

2.1.2 References

2.1.3 Modules

Module: mismatch string kernel Implementattion of mismatch string kernel Author: Meng Zhang <Raina-Meng@outlook.com> Reference:

> Leslie C.S., Eskin E., Cohen A., Weston J., Noble W.S. Mismatch string kernels for discriminative protein classification. Bioinformatics. 2004;20:467–476. doi: 10.1093/bioinformatics/btg431.

<https://papers.nips.cc/paper/2179-mismatch-string-kernels-for-svm-protein-classification.pdf>

class strkernel.mismatch_kernel.**MismatchKernel**(*l=None*, *k=None*, *m=None*, ***kwargs*)

Bases: strkernel.lib.mismatchTrie.MismatchTrie

Python implementation of Mismatch String Kernels. Parameters ———- l: int, optional (default None), size of alphabet.

Examples of values with a natural interpretation: 2: for binary data 256: for data encoded as strings of bytes 4: for DNA/RNA sequence (bioinformatics) 20: for protein data (bioinformatics)

k: int, optional (default None), the k in 'k-mer'. m: int, optional (default None)

maximum number of mismatches for 2 k-mers to be considered 'similar'. Normally small values of m should work well. Plus, the complexity of the algorithm is exponential in m.

****kwargs: dict, optional (default empty)** optional parameters to pass to *tree.MismatchTrie* instantiation.

kernel: 2D array of shape (n_sampled, n_samples), estimated kernel. *n_survived_kmers*: number of leafs/k-mers that survived trie traversal.

get_kernel(*X*, *normalize=True*, ***kwargs*)
Main calling function to get mismatch string kernel.

strkernel.mismatch_kernel.**integerized**(*sequence*)
Convert the character string into numeric string.

strkernel.mismatch_kernel.**normalize_kernel**(*kernel*)
Normalizes a kernel[x, y] by doing: kernel[x, y] / sqrt(kernel[x, x] * kernel[y, y])

strkernel.mismatch_kernel.**preprocess**(*sequences*, *ignoreLower=True*)
Data preprocessing for string sequences. Convert lower case into upper case if 'ignoreLower' is chosen as 'False', else lower case is ignored(default).

Mismatch Trie

2.2 Gappy Kernel

Bio sequences such as DNA, RNA and protein sequences are widely used in SVMs, because they have some certain differences to normal text strings (e.g. a smaller alphabet) kernels fitted to their need were invented. One of them is the gappy-pair kernel[1].

Due to mutations and reading errors in data analysis two genome sequences can be similar even if they do not match exactly but with some gaps. When using the spectrum kernel such similarities will be lost between sequences, therefore the gappy-pair kernel was introduced. Instead of only recognising all k-mers in a sequence for a given k, the gappy-pair kernel also counts how many k-mers with a certain number of gaps(g) appear in the sequence. It does so by redefining k (called k' from here on), which is from now on not referring to the whole length of a k-mer but to half of it, so that the gappy-pair kernel recognises every k-mer of the sort where the first k' bases of the k-mer match the sequence, then a number of gaps from 0 to g is allowed and then the last k' bases match the sequence again. (Therefore, in every k-mer 2k' bases are present.)

For example for the sequence "AACG" the gappy-pair kernel for k'=1 and g=2 would count the following k-mers once: ["AA", "A.C", "A..G", "AC", "A.G", "CG"].

Because it can make sense biologically to not distinguish between a k-mer without gaps and the same k-mer with gaps, our kernel allows the user to decide how to proceed by setting the Boolean variable gapDifferent (Default is set to true). A simple approach of calculating the gappy-pair kernel is to go through all given sequences singularly and then scan through the sequence counting the seen k-mers. To do so it is necessary to create a matrix of the size of all possible k-mers. The number of possible k-mers with gapDifferent being true is (g+1)n^k with n being the length of the alphabet and with gapDifferent being false n^k. Therefore, even if the output is translated into a sparse matrix, during the calculation a significant amount of memory is occupied which can cause problems. A more memory-sufficient

[1] Pavel Kuksa, Pai-Hsi Huang, and Vladimir Pavlovic. A fast, large-scale learning method for protein sequence classification. In 8th Int. Workshop on Data Mining in Bioinformatics, pages 29–37, Las Vegas, NV, 2008.

approach was implemented as well by using a trie structure based on the trie for gapped kernels[2]. Instead of evaluating sequences one by one, all sequences are considered while moving along the trie with a depth-first-search.

2.2.1 How to use the Gappy-Pair Kernel

Define a k' and a g and use either the function gappypair_kernel from gappy_kernel for the simple approach or gappy_trie for the trie approach. Both function return a sparse matrix, which can than be used in a SVM classifier.:

```
from strkernel.gappy_kernel import gappypair_kernel as gk
from strkernel.gappy_trie import gappypair_kernel as gt

sequences = ['ACGTAACT','GTATATAG', 'TAGAGATT']
# Note the t in the input parameters is refering to dna.
# Alternatively, one can choose 1 for rna, 2 for amino acids and 3 for amino acids +␣
↪selenocystein
# simple approach
X1 = gk(sequences,k=1,t=0,g=1)
# trie approach
X2 = gt(sequences,k=1,t=0,g=1)

# For usage in a SVM
from scipy.sparse import coo_matrix, vstack
clf = SVC(C=0.1, kernel='linear', probability=True)
clf.fit(X1, y)
# Or for the trie approach
clf = SVC(C=0.1, kernel='linear', probability=True)
clf.fit(X2, y)
```

2.2.2 References

2.2.3 Modules

Implementation of the gappy kernel.

strkernel.gappy_kernel.**gappypair_kernel**(*sequences*, *k*, *g=0*, *t=0*, *sparse=True*, *reverse=False*, *include_flanking=False*, *gapDifferent=True*)

Compute gappypair-kernel for a set of sequences using k-mer length k and gap size g. The result than can be used in a linear SVM or other classification algorithms. Parameters: ———- sequences: A list of Biopython sequences k: Integer. The length of kmers to consider g: Integer. Gapps allowed. 0 by default. t: Which alphabet according to sequenceTypes.

Assumes Dna (t=0).

sparse: Boolean. Output as sparse matrix? True by default. reverse: Boolean. Reverse complement taken into account?

False by default.

include_flanking: Boolean. Include flanking regions? (the lower-case letters in the sequences given)

gapDifferent: Boolean. If k-mers with different gaps should be threated differently or all the same. True by default.

[2] Leslie C., Kuang R., Fast String Kernels using Inexact Matching for Protein Sequences, Journal of Machine Learning Research, 2004, 1435-1455.

A numpy array of shape (N, 4**k), containing the k-spectrum for each sequence. N is the number of sequences and k the length of k-mers considered.

Implementation of the gappy kernel.

strkernel.gappy_kernel.**gappypair_kernel**(*sequences*, *k*, *g=0*, *t=0*, *sparse=True*, *reverse=False*, *include_flanking=False*, *gapDifferent=True*)

Compute gappypair-kernel for a set of sequences using k-mer length k and gap size g. The result than can be used in a linear SVM or other classification algorithms. Parameters: ———- sequences: A list of Biopython sequences k: Integer. The length of kmers to consider g: Integer. Gapps allowed. 0 by default. t: Which alphabet according to sequenceTypes.

Assumes Dna (t=0).

sparse: Boolean. Output as sparse matrix? True by default. reverse: Boolean. Reverse complement taken into account?

False by default.

include_flanking: Boolean. Include flanking regions? (the lower-case letters in the sequences given)

gapDifferent: Boolean. If k-mers with different gaps should be threated differently or all the same. True by default.

A numpy array of shape (N, 4**k), containing the k-spectrum for each sequence. N is the number of sequences and k the length of k-mers considered.

2.3 Motif Kernel

The Motif Kernel depends on a user defined set of motifs. For each input sequence the number of contained motifs (**motif content**) is computed. These values can then be used as a similarity measurement. Since the features of this kernel are restricted to a user defined set, the performance in later analysis of the kernel output (e.g. classification of cell populations) depends highly on the picked motifs. While this offers a lot of flexibility, the user should carefully decide which motifs are used. The package does not yet include any method to extract significant motifs from a set of sequences. There is, however, a large variety of motif extraction applications available[1][2][3]. Information on how exactly this kernel implementation works can be found in Modules.

A single motif is defined by a sequence of the following elements:

- A single character from an alphabet e.g. [A (Adenine), G (Guanine), T (Thymine), C (Cytosine)]. This character only matches the exact same character.
- The wildcard character **"."** which can represent and therefore matches any of the characters from the alphabet.
- A substitution group, which is a list of characters from the alphabet enclosed in square brackets e.g. **[AG]**. This group matches every character within the brackets (e.g. A or G). If the leading character is a **"^"** the substitution group matches any character **but** those in the brackets (e.g. C or T).

2.3.1 How to use the Motif Kernel

The collection of motifs has to be defined as a list of strings:

[1] Pissis, S.P., Stamatakis, A. and Pavlidis, P., 2013, September. MoTeX: A word-based HPC tool for MoTif eXtraction. In Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics (p. 13). ACM.

[2] Pissis, S.P., 2014. MoTeX-II: structured MoTif eXtraction from large-scale datasets. BMC bioinformatics, 15(1), p.235.

[3] Zhang, Y. and Zaki, M.J., 2006. EXMOTIF: efficient structured motif extraction. Algorithms for Molecular Biology, 1(1), p.21.

```
motif_collection = ["A[^CG]T", "C.G", "C..G.T", "G[A][AT]", "GT.A[CA].[CT]G"]
```

This collection can then be used to create the motif Kernel:

```
from strkernel.motifkernel import motifKernel
motif_kernel = motifKernel(motif_collection)
```

Afterwards a set of sequences can be passed to the motifKernel object which returns a sparse matrix. The sparse matrix contains the motif content for each sequence (default) or the similarities between the sequences based on the motif content (with *return_kernel_matrix* enabled). If the flanking characters (lower case letters) should not be included *include_flanking* has to be disabled. Flanking characters are enabled by default since it generally improves accuracy if the kernel output is used for classification. For large sets of sequences disabling *include_flanking* characters can give a significant performance boost:

```
sequences = ["ACGTCGATGC", "GTCGATAGC", "GCTAGCacgtaCGC","GTAGCTgtgcGTGcgt",
↪"CGATAGCTAGTTAGC"]
#compute motif content matrix
motif_content = motif_kernel.compute_matrix(sequences)
#compute kernel matrix
kernel_matrix = motif_kernel.compute_matrix(sequences, return_kernel_matrix = True)
#compute motif content matrix without considering the flanking characters
matirx_without_flanking = motif_kernel.compute_matrix(sequences, include_flanking =␣
↪False)
```

2.3.2 References

2.3.3 Modules

Motif Kernel Module.

class strkernel.motifkernel.**motifKernel**(*motifs: [<class 'str'>]*)

Bases: object

This is the main class used for the motif kernel construction. The idea is to construct a Trie from a set of Motifs and then use this Trie to compute the motif content of a sequnece. The motif content can then be used to compute the similarity between sequences and therefore is able to serve as input for machine learning algorithms.

Example: An elaborate example can be found in the Tutorials sections.

Standard Use:

```
motifKernel(motifs)
```

compute_matrix(*sequences: [<class 'str'>], include_flanking: bool = True, return_kernel_matrix: bool = False*)

Computes the motif content of a set of sequences and returns a sparse matrix which can be used as input for machine learning approaches. The sparse matrix has only been tested with algorithms from the python package *sklearn*. Optional parameters allow the computation of a kernel matrix and inclusion of flanking regions.

Args: **sequences:** A list of strings that are of the same alphabet as the motifs used to construct the motif Trie.

include_flanking: Option to include or disregard the flanking regions. Default is True.

return_kernel_matrix: A boolean value that indicates if the function should return a sparse matrix with the similarities between sequences (True) or a sparse matrix where each row contains the motif content of a sequence (False). Default is False.

Returns: csr_matrix: A sparse matrix object containg either the kernel matrix (*return_kernel_matrix* = True) or the motif content of each sequence.

Motif Trie

Motif Trie Module

class strkernel.lib.motiftrie.**MotifTrie**(*motifs: [<class 'str'>]*)
Bases: object

The main objective of this class is to construct a Trie from a set of Motifs (strings). The Trie can then be used to compute the motif content of a sequence.

First, Each of the motifs which is passed to the constructer is converted into a Motif object. The Motif objects are then used for the Trie constrcution.

The Trie construction can be seperated into the following steps:

1. The Trie is initialized with a root node (TrieNode object)
2. Each Motif object is added to the Trie by parsing the Trie and adding the parts of the Motif which are not yet present.
3. The final Node object of each Motif is marked.

If sequences are passed to *check_for_motifs* function a DFS is performend and a numpy array containing the motif content is returned.

add(*motif: strkernel.lib.motif.Motif*)
Adds a motif to the Trie.

Args: motif: A motif object that originates from a motif (string) passed to the constructer of the MotifTrie.

check_for_motifs(*sequence: str*) → numpy.array
Iterates over the given sequence and returns the sum of the motif content of all subsequences.

Args: sequence: A sequence (read) that only has characters that also appear in the alphabet of the motifs used to construct the MotifTrie.

Returns: Numpy array containing the motif content of the sequence.

dfs(*sequence: str*, *motifdict: dict*) → [<class 'str'>]
Performs a depth first search on the input sequence and adds the motif content to the input dictionary.

Args: sequence: A part of the sequence given to check_for_motifs. In the first iteration of check_for_motifs the complete sequence is passed to this function. **motifdict:** A dictionary where each entry refers to one of the motifs used to construct the MotifTrie.

Returns: The motifdict with the motif content in this specific sequence.

class strkernel.lib.motiftrie.**TrieNode**(*char: str*)
Bases: object

The TrieNode class consist of an element of a motif and its children in the Motif Trie. This class can be considerd a helper for the MotifTrie class.

Motif

Motif Module.

class strkernel.lib.motif.**Motif**(*motif: str*)
Bases: object

The input string gets transformed into an object of the Motif class. The main attribute of this class is the motif itself which is given as a list of strings where each string is one of the following elements:

1. A character of a common alphabet (e.g. {A,G,C,T}) -> this alphabet should be the same for all motifs
2. The wildcard character "."
3. Substiution groups that contain 2 ore more characters of the alphabet (e.g. [AG] or [CT]). "^" as a leading character indicates that every character in the alphabet but those in the substitution group matches this part of the motif.

get_alphabet()
Extracts the alphabet (unique characters) from a string.

process_motif()
Processes the input motif which is a string and returns a list of strings.

CHAPTER 3

Tutorials

3.1 Motif Kernel / SVM Tutorial

This Tutorial shows how you can combine the motif kernel of the *strkernel* package with a Support Vector Machine (SVM) to predict the cell population based on the motif content of a read sequence.

There are two FASTA files from two different cell populations filled with sequences. If you are not familar with the FASTA format, here is a short explantion. *fibroblast.fa* are sequences obtained from fibroblast while *stemcells.fa* contains sequences from stemcells. The goal of this tutorial is to show that we can use prior knowledge to construct motifs and use those motifs to classify new sequences into the two cell populations. The accuracy of the prediction is in this case secondary as long as we get a result that shows that we can use the motif content of sequences to compare their similarity and therefore classify them.

First, you will need some packages for preprocessing, classification and to plot the results.

[1]:
```
# preprocessing
import numpy as np
from Bio.Seq import Seq
import Bio.SeqIO as sio

# SVM
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# ROC and precision-recall curve
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_precision_
↪score

# plotting
import matplotlib.pyplot as plt  # plotting

# motif kernel
from strkernel.motifkernel import motifKernel
```

3.1.1 Preprocessing

In order to use the data with the SVM provided by *sklearn* we need to read in the FASTA files with *Biopython* and add some labels. In this case we will label the stemcells as positive (1) and the fibroblast as negative (0).

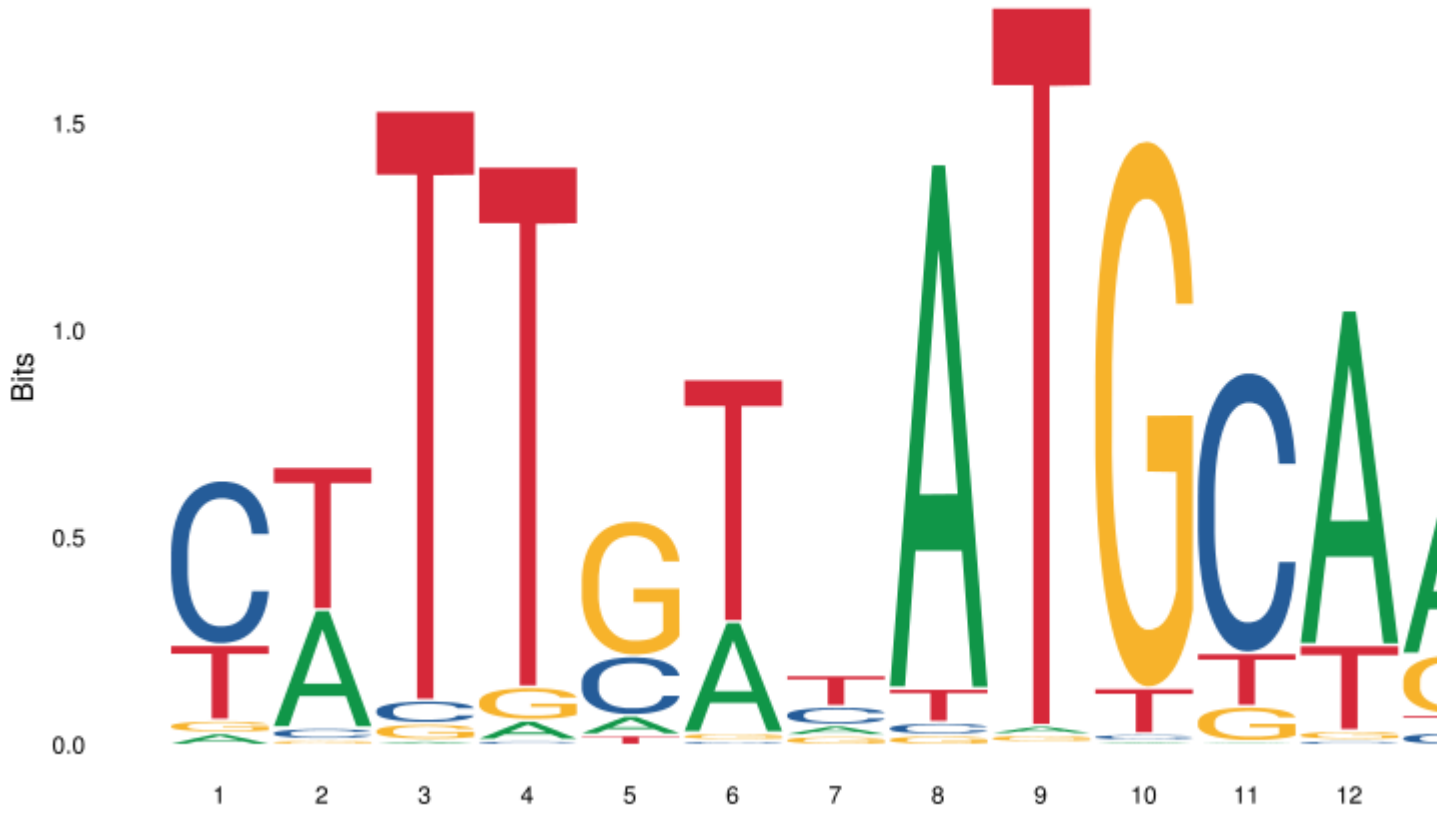
[2]:
```
# load the data
# stemcells
pos_data = [seq.seq for seq in sio.parse('notebook_data/stemcells.fa', 'fasta')]
# fibroblasts
neg_data = [seq.seq for seq in sio.parse('notebook_data/fibroblast.fa', 'fasta')]

pos_labels = np.ones(len(pos_data), dtype=int)
neg_labels = np.zeros(len(neg_data), dtype=int)

y = np.concatenate((pos_labels, neg_labels), axis=0)
```

3.1.2 Motifs

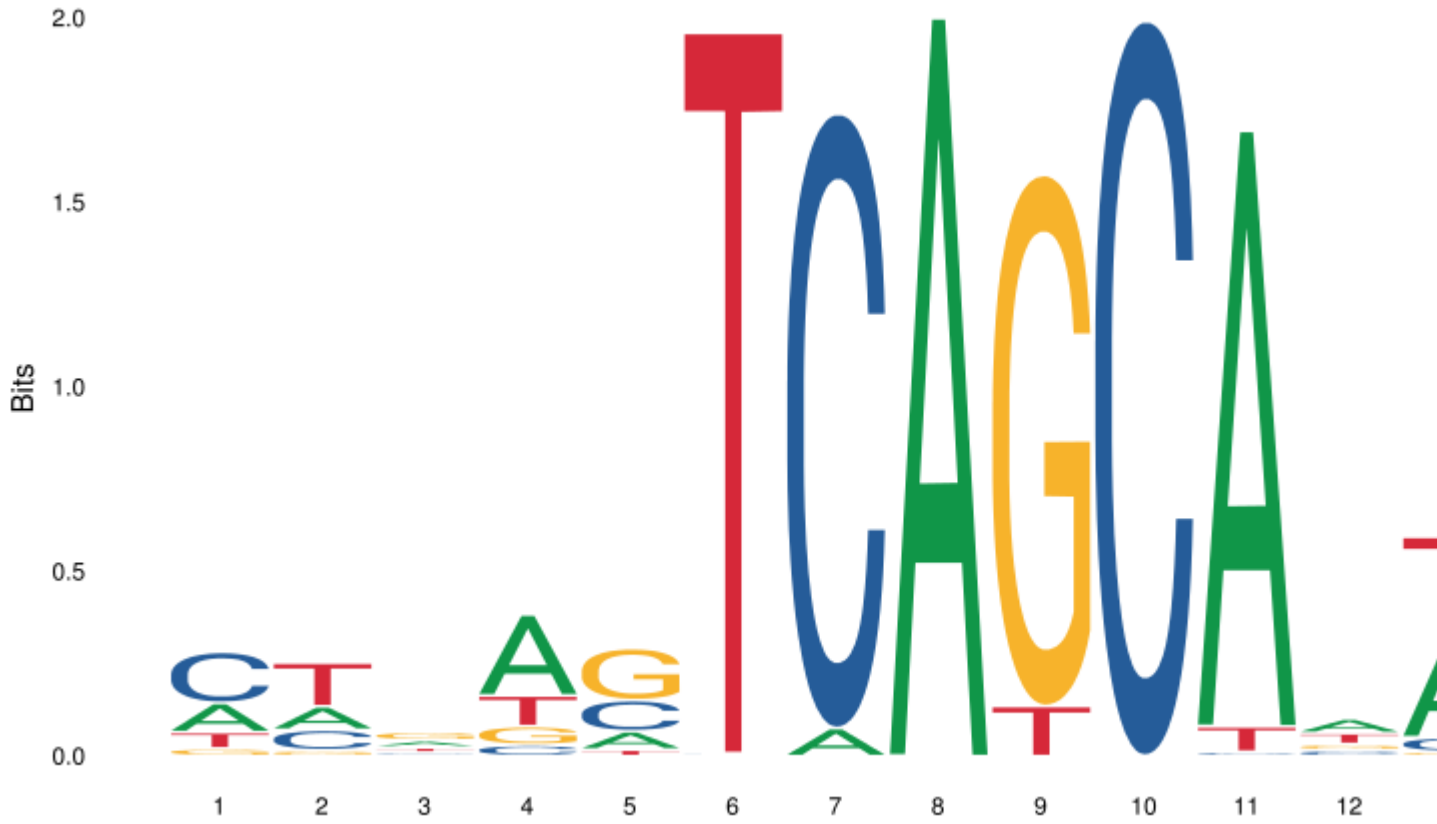
Now we have to decide which collection of motifs we will be using to construct the motif kernel. There are specific transcription factors binding sites which tend to be enriched in one of the groups. Naturally not every sequence contains the binding site for these transcription factors but we should be able to correctly classify those that do. In stemcells, we expect the **oct4** binding site to be enriched because this protein has been shown to be heavily involved in the pluripotency of cells. In fibroblast lung tissue on the other hand, we expect the **mafk** binding site to be enriched.

The jasper database allows to check for the binding motifs of the transcription factors. If we search for **oct4** and **mafk** we get the following binding sites:

oct4



mafk



We can now add the most prominent motifs of these bindings sites to our motif collection. For this tutorial we chose the motif *TCAGCA* of the **mafk** binding site and the motifs *ATGCAA* and *TTGT* of the **oct4** binding site. Since the reads originate from both strands we also have to include the reverse complement of the motifs.

In normal use cases the number of motifs is usally a lot higher but for this tutorial the six motifs will be enough.

[3]:
```
motif_collection = ["TCAGCA","TGCTGA","ATGCAA","TTGCAT","TTGT","ACAA"]

#create the motif kernel
motif_kernel = motifKernel(motif_collection)

#use the motif kernel to compute the motif content matrix for all sequences
motif_matrix = motif_kernel.compute_matrix(pos_data + neg_data)
```

3.1.3 Classification

We can now split the matrix into test and training data. The training data can then be used to train the SVM. We will keep 30% of the data as test data to evaluate the model.

[4]:
```
#split the data into test and training set
X_train, X_test, y_train, y_test = train_test_split(motif_matrix, y, test_size=0.3,␣
↪random_state=42, stratify=y)
```

(continues on next page)

(continued from previous page)

```
#train the classifier
clf = SVC()
clf.fit(X_train, y_train)
```

[4]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

3.1.4 Results

The only thing left to do is to analyze the trained model. *sklearn* provides a function which can be used to produce a classifcation report. This report shows us the precision, recall and f1-score when we apply the model to our test data. We will also plot the ROC and PRC but first we will have to define a couple of wrapper functions.

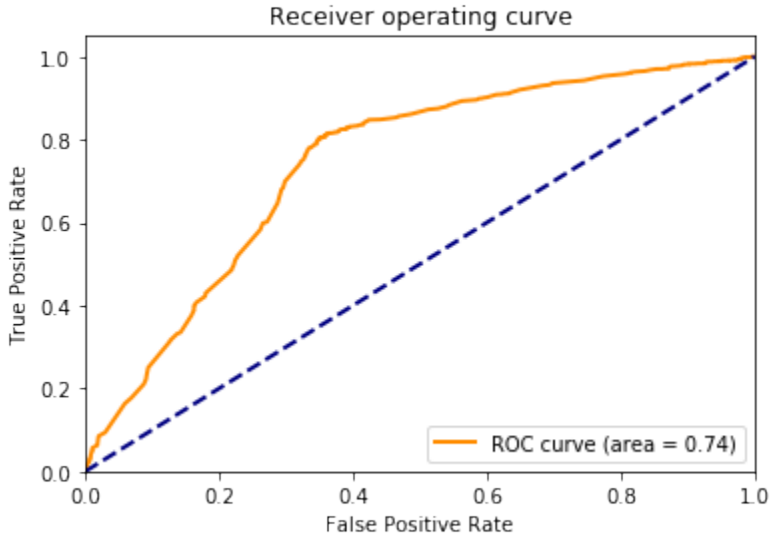
[5]:
```
def plot_roc_curve(y_test, y_score):
    '''Plots a roc curve including a baseline'''
    fpr, tpr, thresholds = roc_curve(y_test, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    lw = 2
    plt.plot(fpr, tpr, color='darkorange',
             lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating curve')
    plt.legend(loc="lower right")
    plt.show()


def plot_prec_recall_curve(y_test, y_scores):
    '''Plots a precision-recall curve including a baseline'''
    precision, recall, thresholds = precision_recall_curve(y_test, y_scores)
    average_precision = average_precision_score(y_test, y_scores)
    baseline = np.bincount(y_test)[1] / sum(np.bincount(y_test))
    plt.figure()
    plt.step(recall, precision, color='b', alpha=0.2,
             where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.2,
                     color='b')
    plt.axhline(y=baseline, linewidth=2, color='navy', linestyle='--')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])
    plt.title('2-class Precision-Recall curve: AP={0:0.2f}'.format(
              average_precision))
    plt.show()


def reportclassfication(clf, X_test, y_test):
```

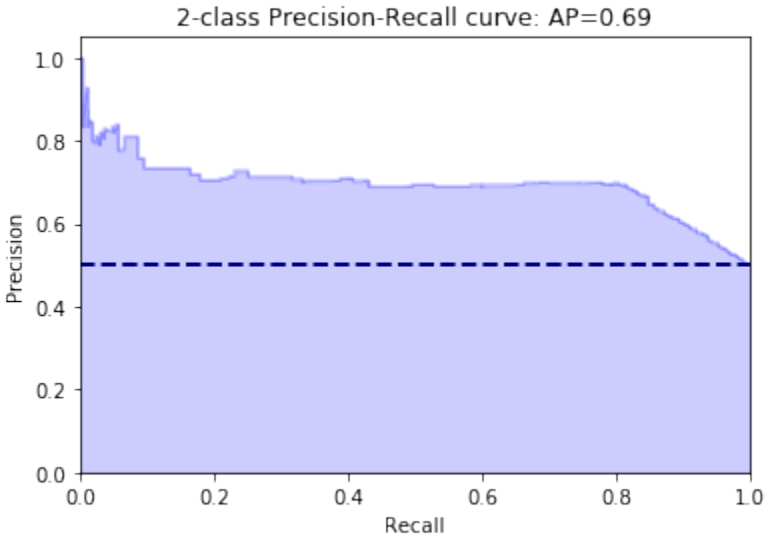
(continues on next page)

(continued from previous page)

```
    '''Reports classification results with the given model and testdata'''

    print("Detailed classification report:")
    print()
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()


# get classfication results with the tuned parameters
reportclassfication(clf, X_test, y_test)

y_scores = clf.decision_function(X_test)
# plot ROC
plot_roc_curve(y_test, y_scores)

# plot PRC
plot_prec_recall_curve(y_test, y_scores)
```

```
Detailed classification report:

             precision    recall  f1-score   support

          0       0.77      0.65      0.70      1200
          1       0.70      0.80      0.75      1199

avg / total       0.73      0.73      0.72      2399
```





We are able to correctly classify around 80% of the stemcell sequences and 70% of the fibroblast sequences. Obviously this is not the best classification results one could have but we can see that the motifs we derived from the bindings sites of transciption factors can be used to classify sequences into cell populations.

In this example we only used the binding sites of two transcription factors to create a motif collection. For better classifcation results you could include more prior information about the cell populations to extend the motif collection.

3.2 Mismatch Kernel / SVM Tutorial

This is an application tutorial of Mismatch String Kernel in package 'strkernel' to Support Vector Machine (SVM) aiming to separate positive and negative sequneces.

In this tutorial we use RNA-Binding protein data (as fasta sequences of the putative binding sites) of proteins IGF2BP123 involved in RNA regulation and known to bind to mRNAs as well as long non-coding RNAs. The original data have been downloaded from the following database http://dorina.mdc-berlin.de, converted to fasta files and pre-processed. Negative or background regions have been generated by taking random genomic regions corresponding to transcribed RNAs, of the same length distribution as the binding site sequences. The data is available in 'notebook_data' directory.

RNA-binding proteins (RBPs) regulate post-transcriptional gene expression and have critical roles in numerous cellular processes including mRNA splicing, export, stability and translation. Despite their ubiquity and importance, the binding preferences for most RBPs are not well characterized. We separate positive and negative sequneces in this tutorial attempting to answer whether RNA-binding protein sites can be discriminated from other RNA regions (or background transcripts) based on sequence alone, and which sequence features are enriched in such binding sites for a particular protein.

First, some python packages are needed to be included, such as self-defined mismatch string kernel, bioinformatics files reading packages and svm related packages.

[6]:
```
from strkernel.mismatch_kernel import MismatchKernel
from strkernel.mismatch_kernel import preprocess

from Bio import SeqIO
from Bio.Seq import Seq
from sklearn.svm import SVC
```

(continues on next page)

(continued from previous page)

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_precision_
↪score
from sklearn.metrics import classification_report  # classfication summary
import matplotlib.pyplot as plt
import numpy as np
from numpy import random
```

3.2.1 Prepare data and preprocessing

As demonstrated above, RBPs data is used in this tutorial. 'Biopython' is a common-used package to read bioinformatics files, for instance, 'fasta' fomat file here. In order to save running time, 2000 positive and negative sequences are randomly opted in our example. The labels of positive sequences are set as 1, negative sequences are set as 0.

In order to use mismatch kernel, sequences are recommended to be preprocessed by calling the function 'preprocess' defined in 'mismatch_kernel'.

[8]:
```
# load the data
posSeq = [seq.seq for seq in SeqIO.parse('notebook_data/positive_IGF2BP123.fasta',
↪'fasta')]
negSeq = [seq.seq for seq in SeqIO.parse('notebook_data/negative_IGF2BP123.fasta',
↪'fasta')]

posX = preprocess(random.choice(posSeq, 2000))
negX = preprocess(random.choice(negSeq, 2000))

# label positive data as 1, negative as 0
posY = np.ones(len(posX), dtype=int)
negY = np.zeros(len(negX), dtype=int)

# compute mismatch kernels used in subsequent SVM training and testing
posKernels = MismatchKernel(l=4, k=5, m=1).get_kernel(posX).kernel
negKernels = MismatchKernel(l=4, k=5, m=1).get_kernel(negX).kernel

# merge data
X = np.concatenate([posKernels,negKernels])
y = np.concatenate([posY,negY])
```

3.2.2 How the kernel looks like – a toy example

Here, I give a toy example to explain how to call the mismatch kernel to compute the kernel and how the mismatch string kernel would look like.

It is obvious that the first two strings in this toy example have high similarity because only the last digits in them are different, so we should expect that similarity estimation of them in the kernel approximately equals to 1, while the estimation between last string and the other two should apparently less than 1.

In 'MismatchKernel' class, we set l=4 since the length of alphabet of the example is 4, just the same as DNA sequences. 'k=3' and 'm=1' present (3,1)-mismatch string kernel, that is, allowing at most 1 mismatch in all 3-mers of sequences.

'leaf_kmers' shows us all mismatch kmers of all sequences and occurence counts of certain kmer in every sequence.

[9]:
```
seq = ['ACGTTGA','ACGTTGT','TCACCGT']
int_seq = preprocess(seq)
```

(continues on next page)

(continued from previous page)

```
mismatch_kernel1 = MismatchKernel(l=4, k=3, m=1).get_kernel(int_seq)
mismatch_kernel2 = MismatchKernel(l=4, k=3, m=1).get_kernel(int_seq, normalize =␣
↪False)
similarity_mat1 = mismatch_kernel1.kernel
similarity_mat2 = mismatch_kernel2.kernel
print(mismatch_kernel1.leaf_kmers)
print(similarity_mat1)
print(similarity_mat2)
```

```
{'001': {2: 2}, '002': {0: 1, 1: 1}, '010': {0: 1, 1: 1, 2: 2}, '011': {0: 1, 1: 1, 2:
↪ 1}, '012': {0: 1, 1: 1, 2: 2}, '013': {0: 1, 1: 1, 2: 1}, '020': {0: 1}, '021': {2:
↪ 1}, '022': {0: 1, 1: 1}, '023': {0: 1, 1: 2, 2: 1}, '031': {2: 1}, '032': {0: 2, 1:
↪ 2}, '033': {0: 1, 1: 1}, '100': {2: 1}, '101': {2: 1}, '102': {2: 2}, '103': {0: 1,
↪ 1: 1, 2: 2}, '110': {2: 2}, '111': {2: 3}, '112': {0: 1, 1: 1, 2: 1}, '113': {0: 1,
↪ 1: 1, 2: 2}, '120': {0: 2, 1: 1, 2: 1}, '121': {0: 1, 1: 1, 2: 2}, '122': {0: 1, 1:
↪ 1, 2: 2}, '123': {0: 1, 1: 2, 2: 1}, '131': {2: 1}, '132': {0: 1, 1: 1, 2: 1}, '133
↪': {0: 2, 1: 2, 2: 1}, '201': {2: 1}, '203': {0: 1, 1: 1}, '210': {2: 1}, '211': {2:
↪ 1}, '212': {0: 1, 1: 1, 2: 1}, '213': {0: 1, 1: 1}, '220': {0: 1}, '223': {0: 2, 1:
↪ 3, 2: 1}, '230': {0: 1, 1: 1}, '231': {0: 1, 1: 1}, '232': {0: 2, 1: 2}, '233': {0:
↪ 1, 1: 1}, '300': {0: 1, 2: 1}, '301': {2: 1}, '302': {0: 1, 1: 1}, '303': {1: 1},
↪'310': {0: 1, 2: 1}, '311': {2: 2}, '312': {0: 2, 1: 2, 2: 2}, '313': {1: 1, 2: 1},
↪'320': {0: 1, 1: 1, 2: 1}, '321': {0: 1, 1: 1}, '322': {0: 2, 1: 2}, '323': {0: 2,␣
↪1: 2, 2: 1}, '330': {0: 2, 1: 1, 2: 1}, '331': {0: 1, 1: 1}, '332': {0: 1, 1: 1},
↪'333': {0: 2, 1: 3}}
[[ 1.          0.92026434  0.48719877]
 [ 0.92026434  1.          0.46153846]
 [ 0.48719877  0.46153846  1.        ]]
[[ 70.  68.  36.]
 [ 68.  78.  36.]
 [ 36.  36.  78.]]
```

As expected, the similarity between the first two strings is arould 1, between last string and the other two is obviously less than 1. In particular, the first kernel matrix is normalized by default and the second one is not by setting the parameter 'normalize' as 'False' in 'get_kernel' function.

3.2.3 Classification

The processed data in last step can be applied to SVM now, before which we need to split it to training and test data. As what is seen following, 30% is used as test in our case.

[10]:
```
# split training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

clf = SVC()
clf.fit(X_train, y_train)
```

[10]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

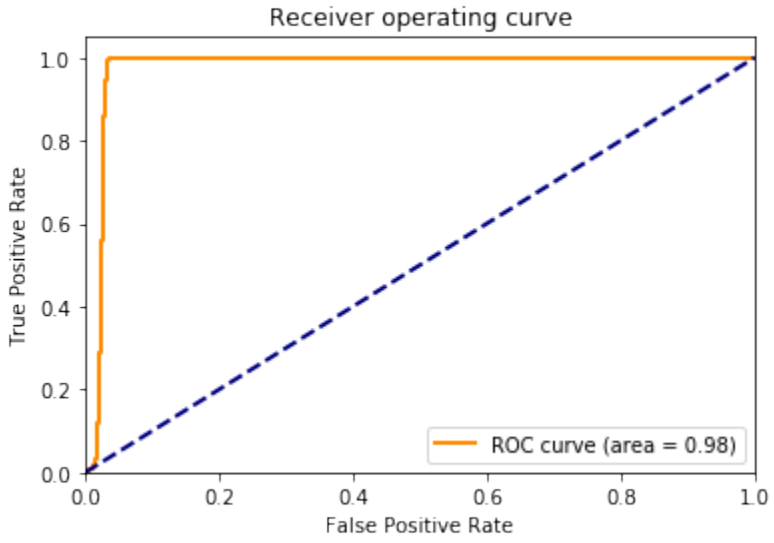
3.2.4 Result

Now, we can check the training result by some functions provided in 'sklearn' package, for instance, 'classification_report' can produce a classifcation report showing us the precision, recall and f1-score of test data. Finally, we

will plot the receiver operating characteristic (ROC) curve and precision recall curve (PRC) to give a more intuitive view.

[11]:
```
y_true, y_pred = y_test, clf.predict(X_test)
print(classification_report(y_true, y_pred))

y_score = clf.decision_function(X_test)

'''Plots a roc curve including a baseline'''
# compute true positive rate and false positive rate
fpr, tpr, thresholds = roc_curve(y_test, y_score)
roc_auc = auc(fpr, tpr)  # compute auc

plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating curve')
plt.legend(loc="lower right")
plt.show()
```

```
             precision    recall  f1-score   support

          0       1.00      0.92      0.96       580
          1       0.93      1.00      0.96       620

avg / total       0.96      0.96      0.96      1200
```

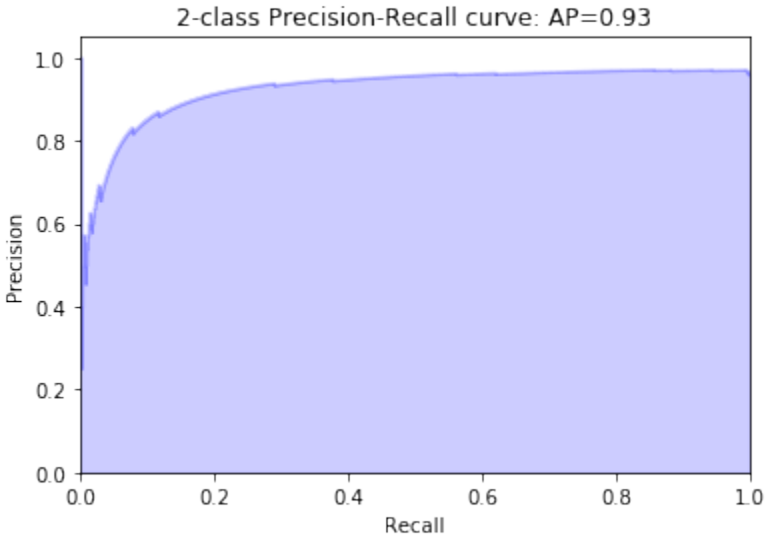


[12]:
```
''' Plot precision and recall curve '''
precision, recall, _ = precision_recall_curve(y_test, y_score)
average_precision = average_precision_score(y_test, y_score)
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2, color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('2-class Precision-Recall curve: AP={0:0.2f}'.format(average_precision))
plt.show()
```



As the above result shows, around 93% of the positive and 100% of the negative sequences are successfully seperated. We would say that, in our case, the mismatch kernel used with an SVM classifier performs very well. One can also try different combinations by changing mismatch key values - 'k', 'm' and SVM model parameters, such as 'C', 'kernel' and 'gamma' etc. to check whether the precision would differ much.

3.3 Gappy Kernel / SVM Tutorial

[1]:
```
import time
from strkernel import gappy_kernel as gk
from strkernel import gappy_trie as gt
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from Bio.Seq import Seq
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve, precision_score
```

[3]:
```
# Reads in files
def read(fname):
    sequences=[]
    with open(fname,'r') as f:
```

(continues on next page)

(continued from previous page)

```
        for line in f:
            if line[0]!='>':
                sequences.append(line.split()[0])
    return sequences

pos=[Seq(x) for x in read('notebook_data/positive_IGF2BP123.fasta')]
neg=[Seq(x) for x in read('notebook_data/negative_IGF2BP123.fasta')]
start = time.time()
k=1
g=2
start = time.time()
spectrum_pos = gk.gappypair_kernel(pos, k=k, g=g,  include_flanking=False,␣
↪gapDifferent = True, sparse = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
spectrum_neg = gk.gappypair_kernel(neg,k=k, g=g,  include_flanking=False,␣
↪gapDifferent = True, sparse = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
spectrum_pos3 = gk.gappypair_kernel(pos, k=k, g=g,  include_flanking=False,␣
↪gapDifferent = False, sparse = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
spectrum_neg3 = gk.gappypair_kernel(neg, k=k, g=g,  include_flanking=False,␣
↪gapDifferent = False, sparse = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
```

```
Calculated 1-gappypair in 12.433693647384644 seconds
Calculated 1-gappypair in 12.717294216156006 seconds
Calculated 1-gappypair in 12.820235013961792 seconds
Calculated 1-gappypair in 12.454049587249756 seconds
```

[4]:
```
from scipy.sparse import coo_matrix, vstack
y = np.concatenate((np.ones(spectrum_pos.shape[0]), -np.ones(spectrum_neg.shape[0])))
X =  vstack([spectrum_pos,spectrum_neg]).toarray()
y3 = np.concatenate((np.ones(spectrum_pos3.shape[0]), -np.ones(spectrum_neg3.
↪shape[0])))
X3 =  vstack([spectrum_pos3,spectrum_neg3]).toarray()
```

[5]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.1,random_state=42,
↪stratify=y)
X_train3, X_test3, y_train3, y_test3 = train_test_split(X3,y3,test_size=0.1,random_
↪state=42,stratify=y3)
```

[6]:
```
start = time.time()
clf = SVC(C=0.1, kernel='linear', probability=True)
clf.fit(X_train, y_train)
print ("Trained linear SVM on {}-spectrum in {} seconds".format(k, time.time() -␣
↪start))
```

```
Trained linear SVM on 1-spectrum in 127.65080642700195 seconds
```

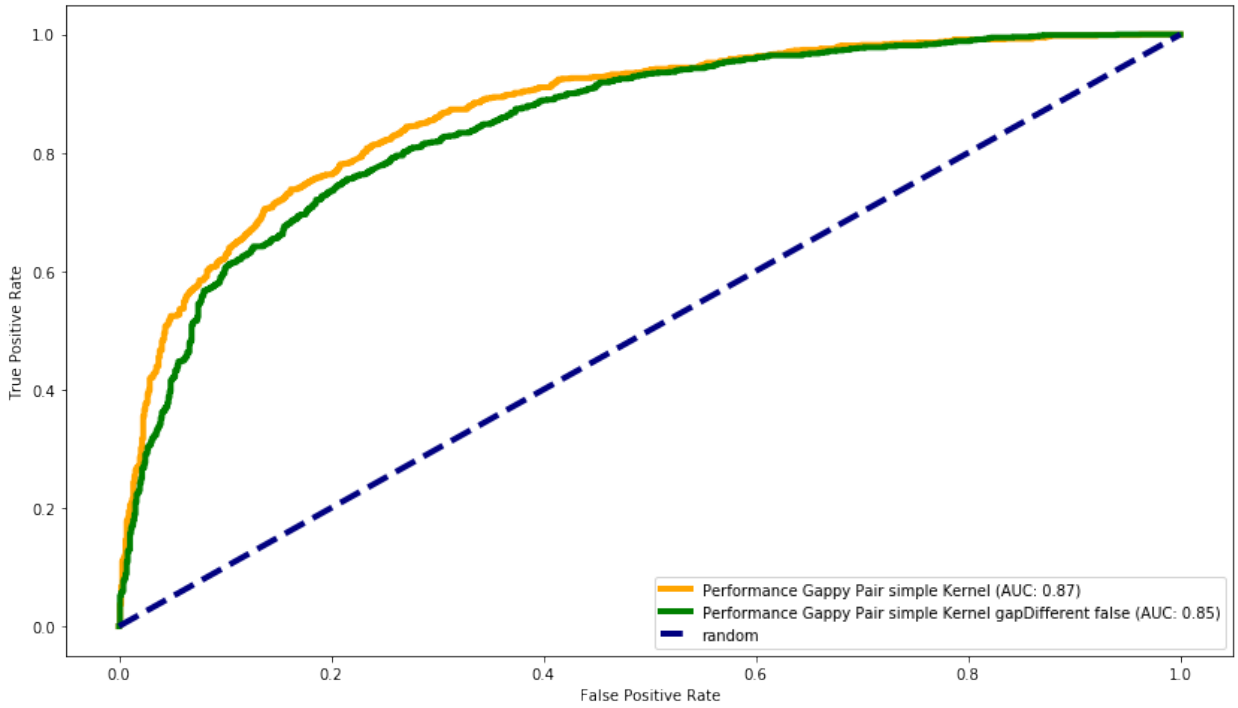
[7]:
```
start = time.time()
clf3 = SVC(C=0.1, kernel='linear', probability=True)
clf3.fit(X_train3, y_train3)
print ("Trained linear SVM on {}-spectrum in {} seconds".format(k, time.time() -␣
↪start))
```

(continues on next page)

(continued from previous page)

```
Trained linear SVM on 1-spectrum in 115.81935048103333 seconds
```

[8]:
```
y_score = clf.predict_proba(X_test)
roc_auc = roc_auc_score(y_score=y_score[:,1], y_true=y_test)
tpr, fpr, _ = roc_curve(y_score=y_score[:,1], y_true=y_test)
y_score3 = clf3.predict_proba(X_test3)
roc_auc3 = roc_auc_score(y_score=y_score3[:,1], y_true=y_test3)
tpr3, fpr3, _ = roc_curve(y_score=y_score3[:,1], y_true=y_test3)
fig = plt.figure(figsize=(14, 8))
plt.plot(tpr, fpr, label='Performance Gappy Pair simple Kernel (AUC: {:0.2f})'.
↪format(roc_auc), lw=4, color='orange')
plt.plot(tpr3, fpr3, label='Performance Gappy Pair simple Kernel gapDifferent false␣
↪(AUC: {:0.2f})'.format(roc_auc3),lw=4, color='green')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.plot([0, 1], [0, 1], color='navy', lw=4, linestyle='--', label='random')
plt.legend(loc='lower right')
fig.savefig('./Gappy_Kernel.pdf')
```



[9]:
```
# Necessary, because the gappy pair trie kernel at the moment can't handle N␣
↪nucleotides
def remove_n(seqs):
    data=[]
    for x in seqs:
        if Seq('N') in x:
            pass
        else:
            data.append(x)
    return data
```

(continues on next page)

(continued from previous page)

```
neg_without_n=remove_n(neg)

start = time.time()
spectrum_pos2 = gt.gappypair_kernel(pos, k, 0, g=g,  include_flanking=False,␣
↪gapDifferent = False)
print ("Calculated {}-gappypair trie in {} seconds".format(k, time.time() - start))
start = time.time()
spectrum_neg2 = gt.gappypair_kernel(neg_without_n, k, 0, g=g,  include_flanking=False,
↪gapDifferent = False)
print ("Calculated {}-gappypair trie in {} seconds".format(k, time.time() - start))
```

```
Calculated 1-gappypair trie in 234.3155553340912 seconds
Calculated 1-gappypair trie in 229.59363436698914 seconds
```

[6]:
```
from scipy.sparse import coo_matrix, vstack
X2 =  vstack([spectrum_pos2,spectrum_neg2]).toarray()
y2 =  np.concatenate((np.ones(spectrum_pos2.shape[0]), -np.ones(spectrum_neg2.
↪shape[0])))
```

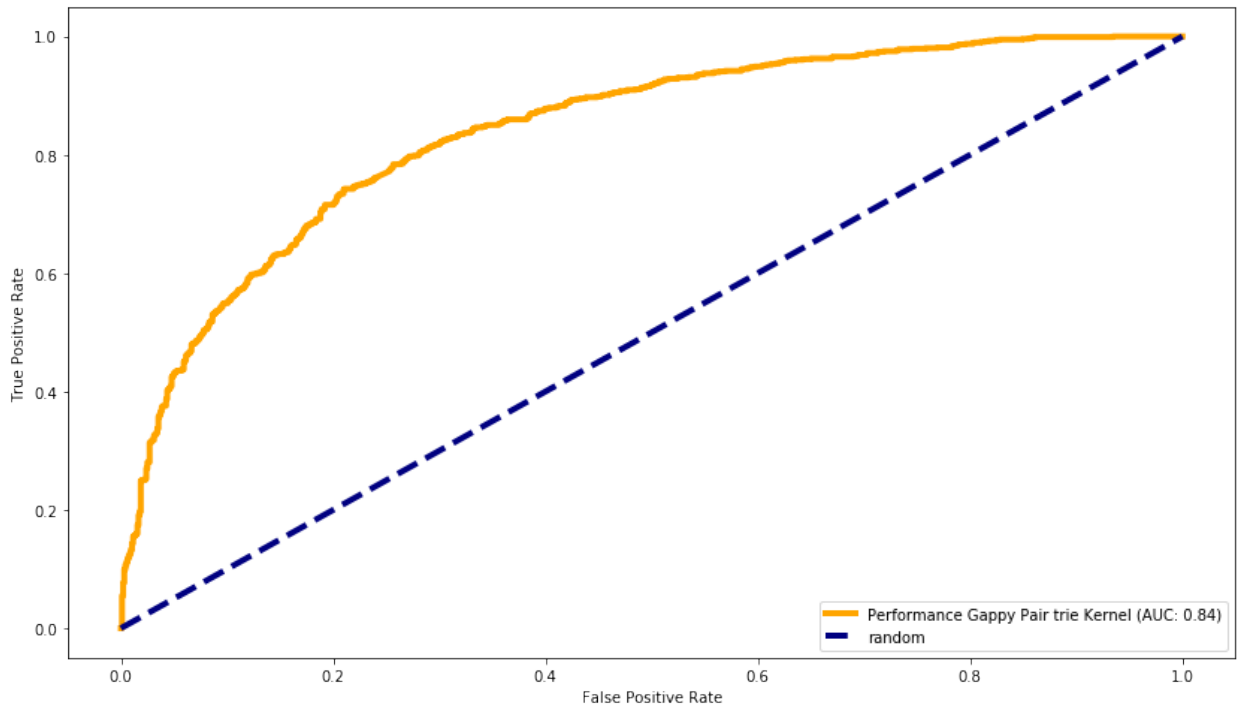
[7]:
```
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2,y2,test_size=0.1,random_
↪state=42,stratify=y2)
```

[8]:
```
start = time.time()
clf2 = SVC(C=0.1, kernel='linear', probability=True)
clf2.fit(X_train2, y_train2)
print ("Trained linear SVM on {}-spectrum in {} seconds".format(k, time.time() -␣
↪start))
```

```
Trained linear SVM on 1-spectrum in 166.927264213562 seconds
```

[19]:
```
y_score2 = clf2.predict_proba(X_test2)
roc_auc2 = roc_auc_score(y_score=y_score2[:,1], y_true=y_test2)
tpr, fpr, _ = roc_curve(y_score=y_score2[:,1], y_true=y_test2)
fig = plt.figure(figsize=(14, 8))
plt.plot(tpr, fpr, label='Performance Gappy Pair trie Kernel (AUC: {:0.2f})'.
↪format(roc_auc2),          lw=4, color='orange')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.plot([0, 1], [0, 1], color='navy', lw=4, linestyle='--', label='random')
plt.legend(loc='lower right')
```

[19]:
```
<matplotlib.legend.Legend at 0x7f63aa9c3fd0>
```



[3]:
```
start = time.time()
gk.gappypair_kernel(pos, k=k, g=g,  include_flanking=False, reverse=True,␣
↪gapDifferent = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
gk.gappypair_kernel(pos,k=k, g=g,  include_flanking=True, gapDifferent = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
gk.gappypair_kernel(pos, k=k, g=g,  include_flanking=False, reverse =True,␣
↪gapDifferent = False)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
gk.gappypair_kernel(pos, k=k, g=g,  include_flanking=True, gapDifferent = False)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
gk.gappypair_kernel(pos, k=k, g=4,  include_flanking=False, gapDifferent = True)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
start = time.time()
gk.gappypair_kernel(pos, k=k, g=4,  include_flanking=False, gapDifferent = False)
print ("Calculated {}-gappypair in {} seconds".format(k, time.time() - start))
```

```
Calculated 1-gappypair in 21.7454092502594 seconds
Calculated 1-gappypair in 242.06059432029724 seconds
Calculated 1-gappypair in 21.176217317581177 seconds
Calculated 1-gappypair in 241.81857538223267 seconds
Calculated 1-gappypair in 32.68045377731323 seconds
Calculated 1-gappypair in 33.1010320186615 seconds
```

[ ]:

[3]:

[41]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

Python Module Index

S

Index