PyML supplement for the Tutorial "SVMs and kernels for computational biology"

Asa Ben-Hur

July 13, 2008

Department of Computer Science
Colorado State University
Fort Collins, CO 80521 USA

Abstract

This document steps the user through the steps of generating the results found in the tutorial **SVMs and kernels for computational biology** using `PyML`.

Contents

1 Introduction

The objective of this supplement is to walk the user through selected parts of the code required for generating the results found in the tutorial using `PyML`. The complete set of results are generated by running the script `pyml_plos.py`. For convenience the commands used in this supplement are found in the file `commands.py`.

In what follows we assume you started your python interpreter in the `pyml_plos` directory, and that you have successfully installed `PyML`. To use `PyML` you need to import it:

```
>>> from PyML import *
```

2 SVMs for Real-Valued Data

Our GC-content data is found in the file 'C_elegans_acc_gc.csv'. We read this comma-delimited file into a dataset object that handles vector data:

```
>>> data = VectorDataSet('C_elegans_acc_gc.csv', labelsColumn = 0)
```

The keyword argument `labelsColumn` indicates to the parser that the labels (+1 or -1, indicating whether an example represents a splice site or not) are found in the first column of the file (indexing follows standard 0-based indexing).

For low dimensional data it is usually a good idea to standardize the data which is performed using the `Standardizer` class, found in the `preproc` module:

```
>>> standardizer = preproc.Standardizer()
>>> standardizer.train(data)  # perform standardization on our dataset
```

We will now instantiate an SVM with a soft margin constant equal to 1:

```
>>> s = SVM(C=1)
```

To perform cross-validation we use the cross-validation method of the SVM we instantiated:

```
>>> results = s.stratifiedCV(data, numFolds=5)
```

Note the use of stratified cross-validation which makes sure that each cross-validation fold contains positive examples and negative examples in the same proportion as in the original dataset. This is important in unbalanced datasets such as our splice-site prediction data. The number of cross-validation folds is provided by the `numFolds` keyword (defaults is 5). The results object returned by the cross-validation method contains information about the accuracy of the model:

```
>>> results
Confusion Matrix:
      Given labels:
       -1   1
   -1 1675  33
    1 325  167
success rate: 0.837273
balanced success rate: 0.836250
area under ROC curve: 0.914589
area under ROC 50 curve: 0.601471
```

More information on results objects is found in the `PyML` tutorial.

To use a nonlinear kernel, e.g. a polynomial kernel we simply attach it to the dataset

```
>>> data.attachKernel('poly', degree = 2)
```

and then perform cross-validation on the data as shown above. For a Gaussian kernel use:

```
>>> data.attachKernel('gaussian', gamma = 0.1)
```

Automatic selection of the SVM soft-margin constant, C and the kernel parameters (the degree of the polynomial kernel or the width of the Gaussian kernel) can be performed automatically using `PyML`'s model selection module (see the `PyML` tutorial for more details).

3 Sequence Kernels

To obtain better performance we now use kernels that directly make use of the sequences flanking the putative splice site. Following the tutorial, we use the spectrum and weighted-degree kernels.

To use the **spectrum kernel** we construct a dataset that represents the spectrum of the set of sequences using the `pyml_plos.py` module:

```
>>> import pyml_plos
>>> data = pyml_plos.gen_spectrum_data('C_elegans_acc.fasta', 1, 5)
```

This generates the spectrum of substrings of lengths 1 to 5. Infact, this method creates a spectrum dataset that differentiates between strings that occur before or after the putative splice site, i.e. within the preceding intron or the following exon. For generating a standard spectrum dataset use the function `PyML.containers.sequenceData.generateSpectrum`. We can now apply the SVM to this dataset:

```
>>> s = SVM(C=1)
>>> results = s.stratifiedCV(data, numFolds=5)
>>> results
Confusion Matrix:
      Given labels:
       -1   1
   -1 1930  55
    1  70  145
success rate: 0.943182
balanced success rate: 0.845000
area under ROC curve: 0.961284
area under ROC 50 curve: 0.775882
```

Our next step is to use the **weighted-degree** kernel. We construct a dataset object from a fasta file by

```
>>> data = SequenceData('C_elegans_acc.fasta', mink = 1, maxk = 3,
                        maxShift=0, headerHandler = pyml_plos.process_header)
```

The weighted degree kernel is the one which is attached by default. We have set it to use strings of lengths between 1 and 3 with no shifts. The `headerHandler` keyword argument provides a function that extracts the sequence ID and label from the header of a fasta file. The PLoS tutorial stresses the importance of normalizing your data. In the case of the weighted degree kernel the appropriate method for normalization is the cosine-like kernel which normalizes kernel values to be less than 1 (see text for details). In `PyML` this is achieved by attaching the `cosine` kernel to the datast:

```
>>> data.attachKernel('cosine')
```

Running the SVM yields a further improvement over the spectrum kernel:

```
>>> s = SVM(C=1)
>>> results = s.stratifiedCV(data, numFolds=5)
>>> results
Confusion Matrix:
      Given labels:
        -1   1
   -1 1906  12
    1  94  188
success rate: 0.951818
balanced success rate: 0.946500
area under ROC curve: 0.987943
area under ROC 50 curve: 0.916765
```