sheet06_thore

June 13, 2019

1 Part A: Kernels for Genes Sequences

In this first exercise, various *degree kernels* such as the weighted degree kernel (WDK) will be implemented. We will use Scikit-Learn (http://scikit-learn.org/) for training SVMs. The focus of this exercise is therefore on the computation of the kernels.

We consider a problem of binary classification of genes sequences. The training and test data is available in the folder `splices-data`. The following code reads the gene sequences data and stores it in numpy arrays.

```
In [1]: import numpy
        Xtrain = numpy.array([numpy.array(list(l)) for l in open('splice-data/splice-train-data.
        Xtest  = numpy.array([numpy.array(list(l)) for l in open('splice-data/splice-test-data.t
        Ttrain = numpy.array([int(l) for l in open('splice-data/splice-train-label.txt','r')])
        Ttest  = numpy.array([int(l) for l in open('splice-data/splice-test-label.txt','r')])
```

1.1 Degree Kernels (20 P)

We consider the degree kernel of degree d applying to two genes sequences x and x' and defined as:

$$k_d(x,x') = \sum_{l=1}^{L-d+1} \mathbf{1}_{u_{l,d}(x)=u_{l,d}(x')}$$

where l iterates over the whole genes sequence, $u_{l,d}(x)$ is a subsequence of x starting at position l and of length d, and $\mathbf{1}_{\{\}}$ is an indicator variable for the equality test given as argument. Given a training set and test set of genes sequences, implement a function that *efficiently* computes the kernel matrices for a certain degree $d \in \{1,2,3,4\}$.

```
In [2]: def getdegreekernels(Xtrain,Xtest,degree):

            ### Replace by our own code
            def getkernel(X, Y):
                N_x, N_y = X.shape[0], Y.shape[0]
                K_c = numpy.zeros((N_x, N_y, X.shape[1]))

                symmetric_calc = None
                xy_equal = numpy.array_equal(X, Y)
                if xy_equal:
```

```
            symmetric_calc = numpy.zeros((X.shape[0], Y.shape[0]))
            symmetric_calc.fill(False)

        # Compare all the rows of X and Y with each other
        if not xy_equal:
            for i in range(N_x):
                x_i = X[i,:]
                for j in range(N_y):
                    K_c[i,j,:] = (x_i == Y[j,:]).astype(int)
        else:
            for i in range(N_x):
                x_i = X[i,:]
                for j in range(N_y):
                    if not symmetric_calc[i,j]:
                        y_j = Y[j,:]
                        comparison = (x_i == y_j).astype(int)
                        K_c[i,j,:] = comparison
                        K_c[j,i,:] = comparison
                        symmetric_calc[i,j] = True
                        symmetric_calc[j,i] = True

        if degree > 1:
            # create a residual copy of K_c
            K_res = numpy.copy(K_c)
            for d in range(1, degree):
                # Sum all the degree next entries
                K_c[:,:,0:-d] += K_res[:,:,d:]
            # Now the K_c values has to cooincide with the degree
            # for a match
            K_c = (K_c == degree).astype(int)

        # Sum up in the third dimension
        return numpy.sum(K_c, axis=2).astype(float)

    Ktrain = getkernel(Xtrain, Xtrain)
    Ktest = getkernel(Xtest, Xtrain)
    ###

    assert(Ktrain.shape==(len(Xtrain),len(Xtrain)) and Ktest.shape==(len(Xtest),len(Xtra
    return Ktrain,Ktest
```

The code below calls the function you implemented for various degrees d, trains SVMs based on these kernels, and measures the prediction accuracy. It can be expected to run in less than 1 minute.

```
In [3]: from sklearn import svm
        Ktrains,Ktests = [None]*4,[None]*4
        for i in range(4):
            Ktrains[i],Ktests[i] = getdegreekernels(Xtrain,Xtest,i+1)
            mysvm = svm.SVC(kernel='precomputed').fit(Ktrains[i],Ttrain)
            Ytrain = mysvm.predict(Ktrains[i])
            Ytest = mysvm.predict(Ktests[i])
            print('degree: %d   training accuracy: %.3f   test accuracy: %.3f'% \
                  (i+1,(Ytrain==Ttrain).mean(),(Ytest==Ttest).mean()))
```

```
degree: 1   training accuracy: 0.994   test accuracy: 0.916
degree: 2   training accuracy: 1.000   test accuracy: 0.934
degree: 3   training accuracy: 1.000   test accuracy: 0.964
degree: 4   training accuracy: 1.000   test accuracy: 0.956
```

1.2 Weighted Degree Kernel (10 P)

We now consider a weighted degree kernel with uniform weights:

$$k(x,x') = \sum_{d=1}^{4} k_d(x,x')$$

where $k_d(x,x')$ is the kernel with degree d that was implemented in the previous section. *Construct* the kernel matrices for the weighted degree kernel and *compute* the training and test accuracy of an SVM trained with this new kernel.

```
In [4]: ### Replace by our own code
        Ktrains,Ktests = getdegreekernels(Xtrain,Xtest,1)

        for i in range(1,4):
            Ktrains_it,Ktests_it = getdegreekernels(Xtrain,Xtest,i+1)
            Ktrains += Ktrains_it
            Ktests += Ktests_it

        mysvm = svm.SVC(kernel='precomputed').fit(Ktrains,Ttrain)
        Ytrain = mysvm.predict(Ktrains)
        Ytest = mysvm.predict(Ktests)

        print('training accuracy: %.3f   test accuracy: %.3f'% \
                  ((Ytrain==Ttrain).mean(),(Ytest==Ttest).mean()))
        ###
```

```
training accuracy: 1.000   test accuracy: 0.967
```

2 Part B: Kernels for Text

Structured kernels can also be used for classifying text data. In this exercise, we consider the classification of a subset of the 20-newsgroups data (available at http://qwone.com/~jason/20Newsgroups/). A subset of this data composed only of texts of class `comp.graphics` and `sci.med` is given in the folder `newsgroup-data`. The first class is assigned label `-1` and the second class is assigned label `+1`. Furthermore, the beginning and the end of the newsgroup messages are removed as they typically contain information that makes the classification problem trivial. Like for the genes sequences dataset, data files are composed of multiple rows, where each row corresponds to one example. The code below extracts the fifth message of the training set and displays its 500 first characters.

```
In [5]: import textwrap
        text = list(open('newsgroup-data/newsgroup-train-data.txt','r'))[4]
        print(textwrap.fill(text[:500]+' [...]'))
```

```
hat is, >>center and radius, exactly fitting those points?  I know how
to do it >>for a circle (from 3 points), but do not immediately see a
>>straightforward way to do it in 3-D.  I have checked some >>geometry
books, Graphics Gems, and Farin, but am still at a loss? >>Please have
mercy on me and provide the solution?   > >Wouldn't this require a
hyper-sphere.  In 3-space, 4 points over specifies >a sphere as far as
I can see.  Unless that is you can prove that a point >exists in
3-space that  [...]
```

2.1 Creating Bag-Of-Words (10 P)

A convenient way of representing text data is as bag-of-words: a set composed of all the words occuring in the document. For the purpose of this exercise, we formally define a word as an isolated sequence of at least three consecutive alphabetical characters. Furthermore, a set of `stopwords` containing mostly uninformative words such as prepositions or conjunctions that should be excluded from the bag-of-word representation is provided in the file `stopwords.txt`. Create a function `text2bow(text)` that converts a text into a bag of words following the just described specifications.

```
In [6]: import re

        def text2bow(text):

            ### Replace by your own code
            words = re.split('[^a-z]+', text.lower())
            words = [item for item in words if len(item) >= 3]
            words = set(words)
            stopwords = set([line.rstrip('\n') for line in open('./stopwords.txt')])
            bow = words - stopwords
            ###

            return bow
```

Your bag-of-words implementation can be tested for the same text shown above by running the code below.

```
In [7]: print(textwrap.fill(str(text2bow(text))))

{'relative', 'could', 'check', 'intersection', 'quite', 'require',
'point', 'bisector', 'lies', 'two', 'fact', 'choose', 'passing',
'call', 'must', 'solution', 'centre', 'meet', 'well', 'possibly',
'center', 'way', 'necessarily', 'lie', 'desired', 'since', 'space',
'close', 'three', 'far', 'books', 'hat', 'exactly', 'equidistant',
'equi', 'surface', 'subject', 'radius', 'coincident', 'immediately',
'least', 'non', 'cannot', 'defined', 'prove', 'geometry', 'gems',
'please', 'plane', 'take', 'know', 'correct', 'fitting', 'happen',
'normally', 'farin', 'yes', 'still', 'say', 'checked', 'distant',
'right', 'graphics', 'failure', 'numerically', 'four', 'pictures',
'points', 'mercy', 'abc', 'perpendicular', 'need', 'circle',
'coplaner', 'angles', 'error', 'let', 'may', 'loss', 'containing',
'collinear', 'define', 'consider', 'otherwise', 'sphere', 'algorithm',
'straightforward', 'steve', 'best', 'bisectors', 'wrong', 'diameter',
'line', 'sorry', 'one', 'unless', 'exists', 'find', 'hyper', 'see',
'provide', 'circumference', 'specifies', 'infinity', 'either',
'normal'}
```

2.2 Implementing Bag-Of-Words Kernels (15 P)

In the following, your task is to implement a simple kernel over bag-of-words. The kernel between two bag-of-words $\mathcal{X}$ and $\mathcal{Y}$ is defined as

$$k(\mathcal{X},\mathcal{Y}) = \sum_{w\in\mathcal{L}} 1_{w\in\mathcal{X}\wedge w\in\mathcal{Y}}$$

where $1_{w\in\mathcal{X}\wedge w\in\mathcal{Y}}$ is an indicator function testing membership to both bags of words. The language $\mathcal{L}$ (set of all existing words) is typically unknown and very large. However, it is computationally equivalent to reduce the language $\mathcal{L}$ to the union $\mathcal{X}\cup\mathcal{Y}$ of the two considered bag-of-words. Thus, we can rewrite the kernel as:

$$k(\mathcal{X},\mathcal{Y}) = \sum_{w\in(\mathcal{X}\cup\mathcal{Y})} 1_{w\in\mathcal{X}\wedge w\in\mathcal{Y}}$$

Create a kernel method that implements this kernel function in a *naive* way. Your naive implementation will then be compared to an optimized one. The naive implementation can be summarized as follows:

- Iterate over all possible words w in $\mathcal{X}\cup\mathcal{Y}$.
- At each iteration test membership of w to $\mathcal{X}$ and $\mathcal{Y}$.
- If both memberships are satisfied, increment the kernel score by 1. If not, leave it to its current value.

Remark: To test the membership of w to $\mathcal{X}$ and $\mathcal{Y}$, do *not* use the operator "`in`" in Python, as it makes use of special data structures behind the scenes. Instead, iterate over all elements of $\mathcal{X}$ and $\mathcal{Y}$ using a `for` loop, and test membership using "==".

```
In [8]: def kernel_naive(bow1,bow2):

            ### Replace by your own code
            L = bow1.union(bow2)
            factor = 0
            for word in L:
                member_bow1 = False
                for candicate in bow1:
                    if candicate == word:
                        member_bow1 = True
                        break

                if member_bow1:
                    for candicate in bow2:
                        if candicate == word:
                            factor += 1
                            break
            return factor
            ###
```

The method `analyze_worstcase_performance(text2bow,kernel)` in `utils.py` computes the worst-case performance (i.e. when applied to the two longest texts in the dataset) of a specific kernel. Run the code below to test the performance of your implementation of the naive kernel.

```
In [17]: import utils
         utils.analyze_worstcase_performance(text2bow,kernel_naive)

kernel score: 761.000 , computation time: 0.440
```

This baseline implementation can be greatly accelerated (by a factor more than 100) by sorting the words in the bag-of-words in alphabetic order, and making use of the new sorted structure in the kernel implementation. In the code below, the sorted list associated to `bow1` is called `sbow1`. *Implement* a function `kernel_sorted(sbow1,sbow2)` that takes as input two lists of words (sorted in alphabetic order) and computes the kernel value in a more efficient manner. Like for the naive implementation, do *not* use the Python operator "`in`".

```
In [10]: def kernel_sorted(sbow1,sbow2):

             ### Replace by your own code
             factor = 0
             last_index = 0

             N1 = len(sbow1)
             N2 = len(sbow2)

             N = N2
             arr1 = sbow1
             arr2 = sbow2
             if N2 < N1:
                 N = N1
                 arr1 = sbow2
                 arr2 = sbow1

             for word in arr1:
                 for i in range(last_index, N):
                     if word == arr2[i]:
                         factor += 1
                         last_index = i + 1
                         break

             return factor
             ###
```

The optimized kernel can be tested for worst case performance by running the code below. Here, we define an additional method `text2sbow(text)` for computing the sorted bag-of-words. Verify that the kernel score remains the same as with the naive implementation. The computation time is expected to drop drastically.

```
In [11]: def text2sbow(text): return sorted(list(text2bow(text)))

         import utils
         utils.analyze_worstcase_performance(text2sbow,kernel_sorted)
```

```
kernel score: 761.000 , computation time: 0.110
```

2.3 Classifying Documents with a Kernel SVM (15 P)

The kernel function between two text documents can be used to build a SVM-based text classifier. Here, we would like to disriminate between the two classes `comp.graphics` and `sci.med` present in the dataset. The code below reads the whole dataset and stores input (mapped to sorted bag-of-words) and labels in the appropriate data structures.

```
In [12]: import numpy
         Xtrain = list(map(text2sbow,open('newsgroup-data/newsgroup-train-data.txt','r')))
         Xtest  = list(map(text2sbow,open('newsgroup-data/newsgroup-test-data.txt','r')))
         Ttrain = numpy.array(list(map(int,open('newsgroup-data/newsgroup-train-label.txt','r'))
         Ttest  = numpy.array(list(map(int,open('newsgroup-data/newsgroup-test-label.txt','r')))
```

As a first step, one needs to build the kernel matrices between pairs of training examples and between training and test examples. After evaluating whether building such matrices is computationally feasible given the performance of your optimized bag-of-words kernel implementation, write the function `build_kernels(Xtrain,Xtest)` for constructing these matrices.

```
In [13]: def build_kernels(Xtrain,Xtest):
             ### Replace by your own code
```

```
    def getkernel(X, Y):
        N_x, N_y = len(X), len(Y)
        K = numpy.zeros((N_x, N_y))

        for i in range(N_x):
            x_i = X[i]
            x_i.sort()
            for j in range(N_y):
                y_j = Y[j]
                y_j.sort()
                K[i,j] = kernel_sorted(x_i, y_j)
        return K
    Ktrain = getkernel(Xtrain, Xtrain)
    Ktest = getkernel(Xtest, Xtrain)
    ###

    assert(Ktrain.shape==(len(Xtrain),len(Xtrain)) and Ktest.shape==(len(Xtest),len(Xtr
    return Ktrain,Ktest
```

These kernel matrices along with the vector of training labels `Ttrain` can be used to train an SVM in the same way as in the previous exercise on genes sequences classification. Write a function that trains an SVM (using scikit-learn with default parameters) and computes the predictions on the training and test data.

```
In [14]: def get_svm_prediction(Ktrain,Ttrain,Ktest):

             ### Replace by your own code
             mysvm = svm.SVC(kernel='precomputed').fit(Ktrain,Ttrain)
             Ytrain = mysvm.predict(Ktrain)
             Ytest = mysvm.predict(Ktest)
             ###

             assert(Ytrain.shape==Ttrain.shape and Ytest.shape==Ttest.shape)
             return Ytrain,Ytest
```

Finally, the functions that you have implemented for classifying the texts can be tested by measuring the training and test accuracy.

```
In [15]: Ktrain,Ktest = build_kernels(Xtrain,Xtest)
         Ytrain,Ytest = get_svm_prediction(Ktrain,Ttrain,Ktest)

         print('training accuracy: %.3f   test accuracy: %.3f'% \
               ((Ytrain==Ttrain).mean(),(Ytest==Ttest).mean()))

training accuracy: 1.000   test accuracy: 0.962
```

Machine Learning 2 Homework 6

Group: DLBSP

June 13, 2019

Excercise 1

a)

```
import numpy as np
for i in range(1000):
    m = np.random.randint(1,20)
    n = np.random.randint(1,20)
    X = np.random.uniform(low=-10,high=10,size=(m,n))
    alpha = np.random.uniform(low=-10,high=10,size=m)
    if alpha.T@(X**2@X.T**2)@alpha < 0:
        print("Not positive semi definite")
        break
else:
    print("Likely positive semi definite")
```

```
OUTPUT: Likely positive semi definite
```

b)

The convolution for two vectors with the same dimension is defined as $(\cdot) * (\cdot) : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}^{2n-1}$, so

$$\begin{aligned} k(x, x') &= \|x * x'\|^2 \\ &= (x * x')^\intercal (x * x') \\ &= \langle x * x', x * x' \rangle \\ &= \langle \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{2n-1} \end{bmatrix}, \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{2n-1} \end{bmatrix} \rangle, \quad c_i \in \mathbb{R} \quad \forall \quad i \in \{1, 2, \cdots, 2n-1\} \\ &= c_1^2 + c_2^2 + \cdots + c_{2n-1}^2 \geq 0 \end{aligned}$$

thus the convolution kernel is positive semi definite.
The Feature map is: $\phi(x) = (x_1^2, x_2^2, ..., x_n^2)^T$

Excercise 2

a)

Firstly we show that the $\alpha_i\alpha_j$-part is positive semi-definite:

$$\sum_{i=1}^{K}\sum_{j=1}^{K}\alpha_i\alpha_j k(x_i,x_j) \not= \Big(\sum_{i=1}^{K}\sum_{j=1}^{K}\alpha_i\alpha_j\Big)\Big(\sum_{i=1}^{K}\sum_{j=1}^{K}k(x_i,x_j)\Big) = \Big(\sum_{i=1}^{K}\alpha_i\Big)^2\Big(\sum_{i=1}^{K}\sum_{j=1}^{K}k(x_i,x_j)\Big)$$

Where $\left(\sum_{i=1}^{K}\alpha_i\right)^2 \geq 0$ is always true for any $\alpha_i \in \mathbb{R}$.
Now for the Kernel:

$$\sum_{i=1}^{K}\sum_{j=1}^{K}k(x_i,x_j) = \sum_{i=1}^{K}\sum_{j=1}^{K}\sum_{m=1}^{M}\beta_m\sum_{n=1}^{N-m+1} I(u_{m,n}(x_i) = u_{m,n}(x_j))$$

Since $\beta_m \geq 0$ is always true, and the indicator function $I \geq 0$ as well, the product of these two real numbers is also positive semi-definite.

So the product of the positive semi-definite α's and the positive semi-definite kernel $k(x_i, x_k)$ is also positive semi-definite.

b)

For the case $M = 1$, the Kernel looks as such:

$$k(x_i,x_j) = \beta_1\sum_{n=1}^{N} I(u_{1,n}(x_i) = u_{1,n}(x_j))$$

The sum above can be replaced with vector product, which means that a valid feature map would be $\phi : x \rightarrow \mathbb{R}^{4N}$:

$$\phi(x) = \sqrt{\beta_1}\begin{bmatrix} I(u_{1,1}(x) = "A") \\ I(u_{1,2}(x) = "A") \\ \vdots \\ I(u_{1,N}(x) = "A") \\ I(u_{1,1}(x) = "C") \\ \vdots \\ I(u_{1,N}(x) = "C") \\ I(u_{1,1}(x) = "T") \\ \vdots \\ I(u_{1,N}(x) = "T") \\ I(u_{1,1}(x) = "G") \\ \vdots \\ I(u_{1,N}(x) = "G") \end{bmatrix}$$

Which means that the first N dimensions of the vector, works as a binary index of all "A"'s, second N dimensions deals with "C"'s, etc.

So that every position in the 4N-Dimensional vector will contribute with +1 only if both strings x_i, x_j are identical at a certain position, when combined in a product, otherwise 0:

$$\langle\phi(x_i), \phi(x_j)\rangle = k(x_i,x_j)$$

for all x_i, x_j

c)

For the case $M = 2, \beta_1 = 0, \beta_2 = 1$, the Kernel looks as such:

$$k(x_i,x_j) = \beta_2\sum_{n=1}^{N-1} I(u_{2,n}(x_i) = u_{2,n}(x_j))$$

Now, since $\beta_1 = 0$, this means that no substrings of length 1 will contribute to the sum, but instead only substrings with length 2. Therefore, we can just extend what the previous kernel could do, but modify it to deal with two characters instead of one:

$$\phi(x) = \sqrt{\beta_2} \begin{bmatrix} I(u_{2,1}(x) = "AA") \\ I(u_{2,2}(x) = "AA") \\ \vdots \\ I(u_{2,N-1}(x) = "AA") \\ I(u_{2,1}(x) = "AC") \\ \vdots \\ I(u_{2,N-1}(x) = "AC") \\ \vdots \\ I(u_{2,N-1}(x) = "GG") \end{bmatrix}$$

Which means that the Dimensions of this vector is $(N-1) \cdot 2^4$, where $(N-1)$ is the number of postions, times the amount of substring permutations 2^4.

And again, in this special case:

$$\langle \phi(x_i), \phi(x_j) \rangle = k(x_i, x_j)$$