Exercises for the course
Machine Learning 2
Summer semester 2021

Abteilung Maschinelles Lernen
Institut für Softwaretechnik und theoretische Informatik
Fakultät IV, Technische Universität Berlin
Prof. Dr. Klaus-Robert Müller
Email: klaus-robert.mueller@tu-berlin.de

Exercise Sheet 5

Exercise 1: Convolution Kernel (20 P)

Let x, x' be two univariate real-valued discrete signals, that we consider in the following to be infinite-dimensional. We consider a discrete convolution over these two signals

$$[x * x']_t = \sum_{\tau=-\infty}^{\infty} x(\tau) \cdot x'(t-\tau)$$

which also produces an infinite-dimensional output signal. We then define the 'convolution kernel' as:

$$k(x,x') = \|x * x'\|^2 = \sum_{t=-\infty}^{\infty} ([x * x']_t)^2.$$

(a) *Show* that the convolution kernel is positive semi-definite, that is, show that

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k(x_i, x_j) \geq 0$$

for all inputs $x_1, \dots, x_N$ and choice of real numbers $c_1, \dots, c_N$.

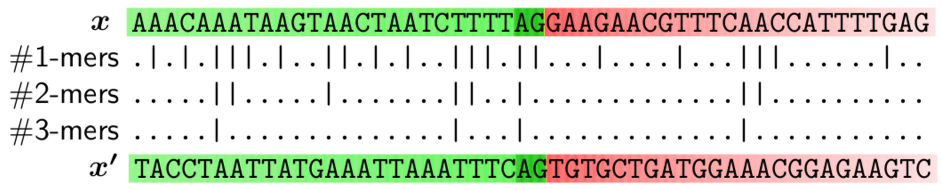
(b) *Give* an explicit feature map for this kernel.

Exercise 2: Weighted Degree Kernels (20 P)

The weighted degree kernel has been proposed to represent DNA sequences ($\mathcal{A} = \{\mathrm{G}, \mathrm{A}, \mathrm{T}, \mathrm{C}\}$), and is defined for pairs of sequences of length L as:

$$k(x,z) = \sum_{m=1}^{M} \beta_m \sum_{l=1}^{L+1-m} I(u_{l,m}(x) = u_{l,m}(z)).$$

where $\beta_1, \dots, \beta_M \geq 0$ are weighting coefficients, and where $u_{l,m}(x)$ is a substring of x which starts at position l and of length m. The function $I(.)$ is an indicator function which returns 1 if the input argument is true and 0 otherwise.


x AAACAAATAAGTAACTAATCTTTTAGGAAGAACGTTTCAACCATTTTGAG
#1-mers .|.|.|||.|..||.|.|..|||.||...|....|...|||......|..
#2-mers .....||.....|.......||..|.............||.........
#3-mers .....|..............|...|.............|.........
x' TACCTAATTATGAAATTAAATTTCAGTGTGCTGATGGAAACGGAGAAGTC

(a) *Show* that k is a positive semi-definite kernel. That is, show that

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k(x_i, x_j) \geq 0$$

for all inputs $x_1, \dots, x_N$ and choice of real numbers $c_1, \dots, c_N$.

(b) *Give* a feature map associated to this kernel for the special case $M = 1$.

(c) *Give* a feature map associated to this kernel for the special case $M = 2$ with $\beta_1 = 0$ and $\beta_2 = 1$.

Exercise 3: Fisher Kernel (20 P)

The Fisher kernel is a structured kernel induced by a probability model $p_\theta(x)$. While it is mainly used to extract a feature map of fixed dimensions from structured data on which a structured probability model readily exists (e.g. a hidden Markov model), the Fisher kernel can in principle also be derived for simpler distributions such as the multivariate Gaussian distribution.

The probability density function of the Gaussian distribution in $\mathbb{R}^d$ of mean μ and covariance Σ is given by:

$$p(x) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\Big(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\Big)$$

In this exercise, we consider the covariance matrix Σ to be fixed, and therefore, the only effective parameter of the model (on which the Fisher kernel is based) is the mean μ.

(a) *Show* that the Fisher kernel associated to this probability model is given by:

$$k(x,x') = (x-\mu)^\top \Sigma^{-1}(x'-\mu)$$

(b) *Give* a feature map associated to this kernel.

Exercise 4: Programming (40 P)

Download the programming files on ISIS and follow the instructions.

Part A: Kernels for DNA Sequences (20 P)

In this first part the weighted degree kernel (WDK) will be implemented for the purpose of classifying DNA sequences. We will use Scikit-Learn (http://scikit-learn.org/ (http://scikit-learn.org/)) for training SVM classifiers. The focus of this exercise is therefore on the computation of the kernels. The training and test data is available in the folder `splices-data`. The following code reads the DNA sequence data and stores it in numpy arrays of characters.

In [1]:
```
import numpy
Xtrain = numpy.array([numpy.array(list(l.rstrip('\r\n'))) for l in open('splice-data/
splice-train-data.txt','r')])
Xtest  = numpy.array([numpy.array(list(l.rstrip('\r\n'))) for l in open('splice-data/
splice-test-data.txt','r')])
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')])
```

We consider the weighted degree kernel described in the lecture. It applies to two genes sequences $x, z \in \{\mathrm{A}, \mathrm{T}, \mathrm{G}, \mathrm{C}\}^L$ and is defined as:

$$k(x,z) = \sum_{m=1}^{M} \beta_m \sum_{l=1}^{L-m+1} I(u_{l,m}(x) = u_{l,m}(z))$$

where l iterates over the whole genes sequence, $u_{l,m}(x)$ is a subsequence of x starting at position l and of length m, and $I(\cdot)$ is an indicator function that returns 1 it the argument is true and 0 otherwise. We would like to implement a function that is capable of *efficiently* computing this weighted degree kernel for any degree M. For this, we will make use of the block method presented in the lecture.

As a first step, we would like to implement a function `size2contrib`, which builds a mapping from a given block size to the kernel contribution, i.e. the sum of beta values associated to all substrings contained in this block. The relation between block size and contribution to the kernel score is as follows:

- Block size 1: contribution = β_1
- Block size 2: contribution = $2\beta_1 + \beta_2$
- Block size 3: contribution = $3\beta_1 + 2\beta_2 + \beta_3$
- etc.

The function should return an integer array of size 101 containing the contribution of blocks of size zero, one, two, up to 100.

In [2]:
```
def size2contrib(beta):

    ## --------------------------------
    ## TODO: Replace by your code
    ## --------------------------------
    import solutions
    s2ctable = solutions.size2contrib(beta)
    ## --------------------------------

    return s2ctable
```

The function can be tested on a simple weighted degree kernel of degree 3 where beta coefficients are given by $\beta_1 = 1, \beta_2 = 3, \beta_3 = 9$.

In [3]:
```
size2contrib(numpy.array([1.0,3.0,9.0]))[:20]
```

Out[3]:
```
array([  0.,   1.,   5.,  18.,  31.,  44.,  57.,  70.,  83.,  96., 109.,
       122., 135., 148., 161., 174., 187., 200., 213., 226.])
```

Having implemented this index, we now focus on implementing the weighted degree kernel. Here, the function `wdk` we would like to implement receives two data arrays `X` and `Z`, and some parameter vector β. The function should return the kernel Gram matrix associated to `X` and `Z`, and run *efficiently*, i.e. as much as possible performing operation over several data points simultaneously by means of array computations. *(Hint: An array of block sizes can be transformed to an array of kernel contributions by using the indexing operation `blockcontribs = s2ctable[blocksizes]` .)*

In [4]:
```
def wdk(X,Z,beta):

    ## -------------------------------
    ## TODO: Replace by your code
    ## -------------------------------
    import solutions
    K = solutions.wdk(X,Z,beta)
    ## -------------------------------

    return K
```

Once the weighted degree kernel has been implemented, the code below trains SVMs on the classification task of interest for different choices of parameters β.

In [5]:
```
from sklearn import svm

for beta in [
    numpy.array([1]),
    numpy.array([1,3,9]),
    numpy.array([0,0,0,1,3,9]),
]:
    Ktrain = wdk(Xtrain,Xtrain,beta)
    Ktest  = wdk(Xtest,Xtrain,beta)
    mysvm = svm.SVC(kernel='precomputed').fit(Ktrain,Ttrain)
    print('beta = %-20s  training: %.3f  test: %.3f'%(beta,mysvm.score(Ktrain,Ttrai
n), mysvm.score(Ktest,Ttest)))
```

```
beta = [1]                 training: 0.994  test: 0.916
beta = [1 3 9]             training: 1.000  test: 0.963
beta = [0 0 0 1 3 9]       training: 1.000  test: 0.933
```

We observe that it is necessary to include non-unigram terms in the kernel computation to achieve good prediction performance. If however, we rely mainly on long substrings, e.g. 4-, 5-, and 6-grams, there are not sufficiently many matchings in the data to obtain reliable similarity scores, and the prediction performance decreases as a result.

Part B: Kernels for Text (20 P)

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 composed only of texts of classes `comp.graphics` and `sci.med`. 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 [6]:
```
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  [...]
```

Converting Texts to a Set-Of-Words

A convenient way of representing text data is as "set-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 set-of-words representation is provided in the file `stopwords.txt` . Create a function `text2sow(text)` that converts a text into a set of words following the just described specifications.

In [7]:
```
def text2sow(text):

    ## -------------------------------
    ## TODO: Replace by your code
    ## -------------------------------
    import solutions
    sow = solutions.text2sow(text)
    ## -------------------------------

    return sow
```

The set-of-words implementation is then tested for the same text shown above:

In [8]:
```
print(textwrap.fill(str(text2sow(text))))
```

```
{'sphere', 'call', 'meet', 'infinity', 'failure', 'hat', 'exactly',
'gems', 'still', 'could', 'collinear', 'there', 'choose', 'and',
'must', 'solution', 'fact', 'line', 'not', 'you', 'close', 'geometry',
'through', 'otherwise', 'algorithm', 'radius', 'this', 'hyper',
'four', 'right', 'please', 'happen', 'unless', 'need', 'two',
'graphics', 'coplaner', 'possibly', 'immediately', 'define',
'relative', 'quite', 'from', 'loss', 'prove', 'those', 'correct',
'provide', 'may', 'they', 'find', 'circle', 'sorry', 'normal',
'either', 'best', 'well', 'cannot', 'such', 'intersection',
'bisector', 'center', 'normally', 'point', 'say', 'numerically',
'let', 'its', 'the', 'consider', 'containing', 'will', 'take',
'centre', 'points', 'any', 'angles', 'see', 'diameter', 'are', 'yes',
'necessarily', 'their', 'plane', 'farin', 'them', 'desired', 'some',
'one', 'checked', 'space', 'way', 'wrong', 'lie', 'wouldn', 'all',
'error', 'non', 'does', 'for', 'bisectors', 'coincident', 'can',
'how', 'which', 'mercy', 'abc', 'distant', 'specifies', 'pictures',
'passing', 'far', 'over', 'equidistant', 'circumference', 'check',
'steve', 'require', 'three', 'have', 'then', 'since',
'straightforward', 'fitting', 'books', 'surface', 'very',
'perpendicular', 'exists', 'defined', 'but', 'lies', 'equi', 'least',
'subject', 'know', 'that'}
```

Implementing the Set-Of-Words Kernel

The set-of-words kernels between two documents x and z is defined as

$$k(x,z) = \sum_{w \in \mathcal{L}} I(w \in x \wedge w \in z)$$

where $I(w \in x \wedge w \in z)$ is an indicator function testing membership of a word to both sets of words. As for the DNA classification exercise, it is important to implement the kernel in an efficient manner.

The function `benchmark(text2sow,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 implementation. Here, the function is tested on some naive implementation of the set-of-words kernel available in `utils.py` .

In [9]:
```
import utils
utils.benchmark(text2sow,utils.naivekernel)
```

```
kernel score: 827.000 , computation time: 0.728
```

The goal of this exercise is to accelerate the procedure by sorting the words in the set-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 `sow1` is called `ssow1`. *Implement* a function `sortedkernel(ssow1,ssow2)` that takes as input two sets of words (sorted in alphabetic order) and that computes the kernel score in an efficient manner, by taking advantage of the sorting structure.

In [10]:
```
def sortedkernel(ssow1,ssow2):

    ## -------------------------------
    ## TODO: Replace by your code
    ## -------------------------------
    import solutions
    k = solutions.sortedkernel(ssow1,ssow2)
    ## -------------------------------

    return k
```

This efficient implementation of the set-of-words kernel can be tested for worst case performance by running the code below. Here, we define an additional method `text2ssow(text)` for computing the sorted set-of-words.

In [11]:
```
def text2ssow(text): return sorted(list(text2sow(text)))

import utils
utils.benchmark(text2ssow,sortedkernel)
```

```
kernel score: 827.000 , computation time: 0.001
```

The kernel score remains the same, showing that our new sorted implementation still produces the same function, however, the computation time has dropped drastically.

Classifying Documents with a Kernel SVM

The set-of-words kernel implemented above can be used to build a SVM-based text classifier. Here, we would like to separate our two classes `comp.graphics` and `sci.med`. The code below reads the whole dataset and stores it in a sorted set-of-words format.

In [12]:
```
import numpy
Xtrain = list(map(text2ssow,open('newsgroup-data/newsgroup-train-data.txt','r')))
Xtest  = list(map(text2ssow,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
'))))
```

Kernel matrices are then produced using our efficient kernel implementation with pre-sorting, and a SVM can be trained to predict the document class.

In [13]:
```
Ktrain = numpy.array([[sortedkernel(ssow1,ssow2) for ssow2 in Xtrain] for ssow1 in Xt
rain])
Ktest  = numpy.array([[sortedkernel(ssow1,ssow2) for ssow2 in Xtrain] for ssow1 in Xt
est])
mysvm = svm.SVC(kernel='precomputed').fit(Ktrain,Ttrain)
print('training: %.3f   test: %.3f'% (mysvm.score(Ktrain,Ttrain),mysvm.score(Ktest,Tt
est)))
```

```
training: 1.000   test: 0.962
```

Exercise Sheet 5

1 Convolutional Kernel

a) $$\sum_{ij}^{N} c_i c_j k(x_i, x_j) \geq 0$$

$$\Leftrightarrow \sum_{ij} c_i c_j \sum_{t=-\infty}^{\infty} (x_i * x_j)_t (x_j * x_i)_t$$

$$\Leftrightarrow \sum_{i=1}^{N} \sum_{j=1}^{N} c_i c_j \sum_{-\infty}^{\infty} \left(\sum_{t=-\infty}^{\infty} x_i(\tau) \cdot x_j(t-\tau)\right)\left(\sum_{-\infty}^{\infty} x_i(\tau') x_i(t-\tau')\right)$$

$$\Leftrightarrow \sum_{s=-\infty}^{\infty} \left(\sum^{N} c_i \sum_{\tau=-\infty} x_i(\tau) x_i(s+\tau)\right)\left(\sum_{j=1}^{N} c_j \sum_{\tau'=-\infty} x_j(\tau') x_j(s+\tau')\right)$$

$$= \sum_{s=-\infty}^{\infty} \left(\sum_{i=1}^{N} c_i \sum_{\tau=-\infty}^{\infty} x_i(\tau) x_i(s+\tau)\right)^2 \geq 0$$

b) $$k(x, x') = \sum_{s} \sum_{\tau} x(\tau) x(s+\tau) \sum_{\tau'} x'(\tau') x'(s+\tau')$$

$$= \langle \underbrace{(x * x)}_{\phi(x)}, \underbrace{(x' * x')}_{\phi(x')} \rangle$$

2 Weighted Degree Kernels

a) $$\sum_{ij} c_i c_j \sum_{m} \beta_m \sum_{l} \mathbb{1}(u_{l,m}(x) = u_{l,m}(z)) \geq 0$$

$$\Leftrightarrow \sum_{ij} c_i c_j \sum_{m,l} \beta_m \sum_{s \in A^m} \mathbb{1}(u_{l,m}(x) = s)\, \mathbb{1}(u_{l,m}(z) = s)$$

$$\Leftrightarrow \sum_{l,m,s} \beta_m \sum_i c_i \mathbb{1}(u_{l,m}(x) = s) \sum_j c_j \mathbb{1}(u_{l,m}(z) = s)$$

$$\Leftrightarrow \sum_{l,m,s} \underbrace{\beta_m}_{\geq 0} \underbrace{\left(\sum_i c_i \mathbb{1}(u_{l,m}(x = s))\right)^2}_{\geq 0} \geq 0$$

b) $$\sum_{m,l} \beta_m \sum_{s\in A^m} \mathbb{1}(u_{l,m}(x)=s)\,\mathbb{1}(u_{l,m}(z)=s)$$

$$\Leftrightarrow \sum_{l,s} \sqrt{\beta}\,\mathbb{1}(u_l(x)=s)\,\sqrt{\beta}\,\mathbb{1}(u_l(z)=s)$$

$$\Leftrightarrow \langle \underbrace{\sqrt{\beta}\,\mathbb{1}(u_l(x)=s)_{l,s}}_{\phi(x)}, \underbrace{\sqrt{\beta}\,\mathbb{1}(u_l(z)=s))_{l,s}}_{\phi(z)} \rangle$$

c) $$k(x,x') = \sum_l \underbrace{\beta_2}_{=1} \sum_{s\in A^2} \mathbb{1}(u_{l2}(x)=s)\,\mathbb{1}(u_{l2}(x')=s)$$

$$= \langle (\mathbb{1}(u_{l2}(x)=s))_{l,s}, (\mathbb{1}(u_{l2}(x')=s))_{l,s} \rangle$$

3 Fisher Kernel

a) $$k(x,x') = G_x^T E_z[G_z G_z^T]^{-1} G_{x'}$$

$$=(\Sigma^{-1}(x-\mu))^T E_z[(\Sigma^{-1}(x-z))(\Sigma^{-1}(x-z))^T]^{-1}$$
$$\Sigma^{-1}(x'-\mu)$$

$$= (x-\mu)^T \Sigma^{-1} (\Sigma^{-1} \underbrace{\Sigma\,\Sigma^{-1}}_{\mathbb{1}})^{-1} \Sigma^{-1} (x'-\mu)$$

$$= (x-\mu)^T \Sigma^{-1} \underbrace{(\Sigma^{-1})^{-1} \Sigma^{-1}}_{=\mathbb{1}} (x'-\mu)$$

$$= (x-\mu)^T \Sigma^{-1} (x'-\mu)$$

b) $$k(x,x') = (x-\mu)^T \Sigma^{-1} (x'-\mu)$$

$$= (x-\mu)^T L L^T (x'-\mu)$$

$$= (L^T(x-\mu))^T (L^T(x'-\mu)) = \langle \underbrace{L^T(x-\mu)}_{\phi(x)}, \underbrace{L^T(x'-\mu)}_{\phi(x')} \rangle$$