Machine Learning 2 Homework 7

Group: DLBSP

June 27, 2019

Exercise 1

a)

$x \in \mathbb{R}^d$

$$\sum_i^d \sum_j^d \alpha_i \alpha_j k(x_i, x_j) = \sum_i^d \sum_j^d \alpha_i \alpha_j \sum_l^L \beta_l k_l(x_i, x_j) = \sum_l^L \beta_l \sum_i^d \sum_j^d \alpha_i \alpha_j k_l(x_i, x_j)$$

Since k_l is a kernel

$$\sum_i^d \sum_j^d \alpha_i \alpha_j k_l(x_i, x_j) \geq 0$$

and thus if $\beta_1, \cdots \beta_L \geq 0$ then

$$\sum_i^n \sum_j^n \alpha_i \alpha_j k(x_i, x_j) = \sum_l^L \beta_l \sum_i^d \sum_j^d \alpha_i \alpha_j k_l(x_i, x_j) \geq 0$$

and k is positive semi-definite.

b)

$$\begin{aligned} k(x, x') = \sum_l^L \beta_l k_l(x, x') &= \sum_l^L \beta_l \Phi_l(x)^\intercal \Phi_l(x') \\ &= \begin{bmatrix} \sqrt{\beta_1}\Phi_1^\intercal(x) & \cdots & \sqrt{\beta_L}\Phi_L^\intercal(x) \end{bmatrix} \begin{bmatrix} \sqrt{\beta_1}\Phi_1(x') \\ \vdots \\ \sqrt{\beta_L}\Phi_L(x') \end{bmatrix} \end{aligned}$$

Thus

$$\Phi(x) = \begin{bmatrix} \sqrt{\beta_1}\Phi_1(x) \\ \vdots \\ \sqrt{\beta_L}\Phi_L(x) \end{bmatrix}$$

Exercise 2

a)

$x \in \mathbb{R}^n$, $y \in \{1, \cdots, C\}$ and $C \in \{n \in \mathbb{N} : n \geq 2\}$

$$\sum_i^d \sum_j^d \alpha_i \alpha_j k(x_i, x_j) \cdot \mathbb{1}_{y=y} = \sum_i^d \sum_j^d \alpha_i \alpha_j k(x_i, x_j) \geq 0$$

since k is a kernel.

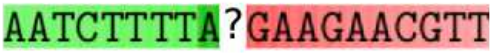
b)

sheet08

June 27, 2019

1 Structured Output Learning

In this exercise, we consider a data completion task to be solved with structured output learning. The dataset is based on the dataset of the previous programming sheet on splice sites classification. We would like to be able to reconstruct a nucleotide sequence when one of the nucleotides is missing. One such incomplete sequence of nucleotides is shown in the image below



where the question mark indicates the missing nucleotide. We would like to make use of the degree kernel that was used in the previous programming sheet. It was shown to represent genes data efficiently near the splice sites. For our completion task, we adopt a structured output learning approach, where the candidate value for replacing the missing nucleotide is also part of the kernel feature map. Interestingly, with this approach, the kernel can still apply to the same type of input data (i.e. continuous gene sequences) as in the standard splice classification setting.

The structured output problem is defined as solving the soft-margin SVM optimization problem:

$$\min_{w,b} \|w\|^2 + C\sum_{i=1}^{N} \xi_i$$

where for all inputs pairs $(x_i, y_i)_{i=1}^N$ representing the genes sequences and the true value of the missing nucleotide, the following constraints hold:

$$\begin{aligned} w^\top \phi(x_i, y_i) + b &\geq 1 - \xi_i \\ \forall z_i \in \{\mathrm{A,T,C,G}\} \setminus y_i : \; w^\top \phi(x_i, z_i) + b &\leq -1 + \xi_i \\ \xi_i &\geq 0 \end{aligned}$$

Once the SVM is optimized, a missing nucleotide y for sequence x can be predicted as:

$$y(x) = \arg\max_{z \in \{\mathrm{A,T,C,G}\}} w^\top \phi(x, z).$$

The feature map $\phi(x, z)$ is implicitely defined by the degree kernel between gene sequences r and r' given as

$$k_d(r, r') = \sum_{i=1}^{L-d+1} 1_{\{r[i \ldots i+d] = r'[i \ldots i+d]\}}$$

where r is built as the incomplete genes sequence x with missing nucleotide "?" set to z, and where $r[i \ldots i+d]$ is a subsequence of r starting at position i and of length d.

1.1 Loading the Data

The following code calls a function from the file `utils.py` that loads the data in the IPython notebook. Note that only the 20 nucleotides nearest to the splice site are returned. The code then prints the first four gene sequences from the dataset, where the character "?" denotes the missing nucleotide. The label associated to each incomplete genes sequences (i.e. the value of the missing nucleotide "?") is shown on the right.

```
In [1]: import utils
        Xtrain,Xtest,Ytrain,Ytest = utils.loaddata()

        print("".join(Xtrain[0])+"  ?="+Ytrain[0])
        print("".join(Xtrain[1])+"  ?="+Ytrain[1])
        print("".join(Xtrain[2])+"  ?="+Ytrain[2])
        print("".join(Xtrain[3])+"  ?="+Ytrain[3])
```

```
CAACGATCCAT?CATCCACA  ?=C
CAGGACGGTCA?GAAGATCC  ?=G
AAAAAGATGA?GTGGTCAAC  ?=A
TGTCGGTTA?CAATGATTTT  ?=C
```

It can be observed from the output that the missing nucleotide is not always at the same position. This further confirms that the problem cannot be treated directly as a standard multiclass classification problem. Note that in this data, we have artificially removed nucleotides in the training and test set so that we have labels y available for training and evaluation.

1.2 Generating SVM Data (20 P)

In the SVM structured output formulation, the data points (x_i, y_i) denote the true genes sequences and are the SVM positive examples. To be able to train the SVM, we need to generate all possible examples $((x_i, z_i))_{z_i \in \{\text{A,T,C,G}\}}$.

Your first task is to implement a function `builddata(X,Y)` that receives as input the dataset of size `(N x L)` of incomplete gene sequences `X` where `N` is the number of gene sequences and `L` is the sequence length, and where `Y` of size `N` contains the values of missing nucleotides.

Your implementation should produce as output an extended dataset of size `(4N x L)`. Also, the function should return a vector of labels `T` of size `4N` that is `+1` for positive SVM examples and `-1` for negative SVM examples. For repeatability, ensure that all modifications of the same gene sequence occur in consecutive order in the outputs `XZ` and `T`.

```
In [64]: import numpy as np

         def builddata(X,Y):

             # Replace by your own code
             (N,L) = X.shape
             XZ = np.chararray((X.shape[0]*4, L)).astype(str)
             T = np.zeros(len(Y)*4)
             Y = np.array(Y)
```

```
    replacements = ['A', 'T', 'C', 'G']
    for i in range(len(replacements)):
        X_c = np.copy(X)
        X_c[X_c == '?'] = replacements[i]
        XZ[np.arange(i, 4*N, step=4),:] = X_c

        T_wc = np.ones(N)
        T_wc[Y != replacements[i]] = -1
        T[np.arange(i, 4*N, step=4)] = T_wc
    # ---

    assert(len(XZ)==len(T)==4*len(X)==4*len(Y))

    return XZ,T
```

Your implementation can be tested by running the following code. It applies the function to the training and test sets and prints the first 12 examples in the training set (i.e. all four possible completions of the first three gene sequences).

```
In [65]: XZtrain,Ttrain = builddata(Xtrain,Ytrain)
         XZtest,_       = builddata(Xtest ,Ytest )

         for xztrain,ttrain in zip(XZtrain[:12],Ttrain[:12]):
             print("".join(xztrain)+' %+1d'%ttrain)
```

```
CAACGATCCATACATCCACA -1
CAACGATCCATTCATCCACA -1
CAACGATCCATCCATCCACA +1
CAACGATCCATGCATCCACA -1
CAGGACGGTCAAGAAGATCC -1
CAGGACGGTCATGAAGATCC -1
CAGGACGGTCACGAAGATCC -1
CAGGACGGTCAGGAAGATCC +1
AAAAAGATGAAGTGGTCAAC +1
AAAAAGATGATGTGGTCAAC -1
AAAAAGATGACGTGGTCAAC -1
AAAAAGATGAGGTGGTCAAC -1
```

1.3 SVM Optimization and Sequences Completion (30 P)

In this section, we would like to create a function that predicts the missing nucleotides in the gene sequences. The function should be structured as follows: First, we build the kernel training and test matrices using the function `utils.getdegreekernels` and using the specified `degree` parameter. Using `scikit-learn` SVM implementation (`sklearn.svm.SVC`) to train the SVM associated to the just computed kernel matrices and the target vector `Ttrain`. Use the default SVM hyperparameter `C=1` for training.

After training the SVM, we would like to compute the predictions for the original structured output problem, that is, for each original gene sequence in the training and test set, the choice of missing nucleotide value for which the SVM prediction value is highest. The outputs `Ptrain` and `Ptest` denote such predictions and should be arrays of characters `A,T,C,G` of same size as the vectors of true nucleotides values `Ytrain` and `Ytest`.

(Hint: You should consider that in some cases there might be not exactly one missing nucleotide value that produces a positive SVM classification. In such cases, we would still like to find the unique best nucleotide value based on the value of the discriminant function for this particular data point. A special function of scikit-learn's `SVC` class exists for that purpose.)

```
In [ ]: def predict(XZtrain,XZtest,Ttrain,degree):

            # Replace by your own code
            import solutions
            Ptrain,Ptest = solutions.predict(XZtrain,XZtest,Ttrain,degree)
            # ---

            return Ptrain,Ptest
```

The code below tests the prediction function above with different choices of degree parameters for the kernel. Note that running the code can take a while (up to 3 minutes) due to the relatively large size of the kernel matrices. If the computation time becomes problematic, consider a subset of the dataset for development and only use the full version of the dataset once you are ready to produce the final version of your notebook.

```
In [ ]: import solutions

        for degree in [1,2,3,4,5,6]:

            Ptrain,Ptest = predict(XZtrain,XZtest,Ttrain,degree)

            acctr = (Ytrain==Ptrain).mean()
            acctt = (Ytest ==Ptest ).mean()

            print('degree: %d  train accuracy: %.3f  test accuracy: %.3f'%(degree,acctr,acctt))
```