Machine learning with kernel methods 2021
Challenge data Report
Team Name : NOYAU DUR
MVA

NAIT SAADA Thiziri
Télécom Paris
thiziri.naitsaada@telecom-paris.fr

TORRES PÉREZ Claudia
Télécom Paris
claudia.torresperez@telecom-paris.fr

Abstract

This report briefly sums up our main approaches and concerns in tackling the data challenge competition that is hosted on Kaggle. All the implementation details can be found in our Github repository https://github.com/claudiatorresp/kernel-challenge.git.

1. Data

1.1. Introduction

We were provided two types of datasets : numeric data and sequences data. For each of them, three datasets are available and we will try to learn the underlying distributions of each of them. Strictly speaking, we will present in the end one model per dataset and we expect the hidden/testing data of each dataset to come from the same distribution as the ones we were given so that our model has a chance to perform predictions well. It is a binary classification task (labels 0 or 1).

1.2. Train - Validation split

In both cases of numeric or sequences data, we decided to split the data provided into two parts : 80% of training data, 20 % of validation. The first part will be used to train the models while the second one will help with tuning parameters.

2. Kernels used

2.1. Kernels on numeric data

We were provided numeric data, on which it might be worth trying some standard kernels. It was an opportunity for us to do a sanity check of our classifiers implementation.

2.1.1 Linear and Polynomial kernels

We had to play with the degree of the polynomial function we were handling :

$$K(x, y) = \langle x, y \rangle^p$$

2.1.2 Gaussian kernel

It is maybe the most popular approach in kernel methods. The variance had be to tuned to perform the best classification :

$$K(x, y) = e^{\frac{-||x-y||^2}{2\sigma^2}}$$

2.2. Kernels on sequences data

As recommended in the challenge description, we also wanted to manipulate the biological sequences, for which it was necessary to implement kernels that could handle them.

2.2.1 Spectrum kernel

If we denote by $\phi_u(x)$ the number of occurrences of a substring u of length k in a sequence x, then the k-spectrum kernel is given by:

$$K(x, x') = \sum_{u \in \mathcal{A}^k} \phi_u(x)\phi_u(x')$$

To implement the spectrum kernel, we first had to compute the dictionary $\mathcal{A}^k$ which contains all the possible substrings of length k with the letters A, C, G and T. Then, we had to compute the embedding of each sequence, $\phi(x) \in \mathbb{R}^{4^k}$. We decided to create a dictionary, where each key corresponds to a possible substring and the corresponding value is its index in the embedding vector. Hence, to compute the embedding of a sequence, one just has to refer to the dictionary. To compute the kernel matrix, we only have to perform dot products on the embedding vectors, which is done rapidly with sparse matrices.

Using these two structures made our implementation of the spectrum kernel extremely fast, with less than 20 seconds being necessary to compute all the kernels on the training set when $k = 8$.

2.2.2 Mismatch kernel

This kernel is highly similar to the spectrum kernel, except that it allows for mismatches on the computation of the embedding. Its implementation is highly similar to the one of the spectrum kernel, only the dictionary differs. Each key corresponds to a substring, and has for values its index and those of its neighboring substrings (those that are less than m-mismatches away).

3. Classifiers

In order to perform the classification, task, we first need to find a vector $\alpha \in \mathbb{R}^n$ such that $(\hat{f}(x_1), \ldots, \hat{f}(x_n))^T = K\alpha$, where $\hat{f}$ is the minimiser of a certain penalized loss function that depends on the chosen classifier.
We implemented **Kernel Ridge Regression**, **Kernel Logistic Regression** and **SVM**. In order to find their corresponding α, we used both numpy and cvxopt tools. For all three methods, the only parameter that can be tuned is the regularization term λ.

4. Results

Given some primary results, we used traditional techniques in order to boost them.

4.1. Fine Tuning

We performed a series of training - predictions on each of the three validation sets in order to find the parameters/models that would lead to the best performance. It is often called Grid Search. Please find in 2 some of them. The purpose was then to keep the parameters that maximized the accuracy on the validation set for each model. Then, given all these models, the final step was to choose the best one according to its performance on validation set.

4.2. Majority voting

Given a bunch of models with optimized parameters, we wanted to get the most of each of them. So we decided to implement an ensemble method that would combine different predictions coming from different models. More precisely, each model forecasting a class, the final prediction is the class that has received a majority of votes from all these models. This method has improved our results.

4.3. Boosting

One drawback of the previous ensemble method is that each SVM classifier has equal weight, whereas we could decide to trust each SVM classifier depending on a weight. This method is called Weighted Majority voting. However, we were not completely satisfied with such an approach. We wanted to weight each classifier with a weight that depends on its classification performance. To do so, we kept the probability output of belonging to the positive class (instead of a binary output) and we weighted it with a factor reflecting how much we could trust each classifier. In the end, we thresholded this probability and output the class associated (+1 if probability above 0, -1 otherwise).

Our choice of each classifier weight is inspired from AdaBoost algorithm. Given a SVM classifier i, with training error err_i, its associated weight w_i is given by :

$$w_i = \log(\frac{1 - err_i}{err_i})$$

Somehow, it actually corresponds to performing a single iteration of AdaBoost algorithm.

4.4. Final Leaderboard Result

Our selected submissions both used the mismatch kernels with only one mismatch. For dataset 0 and 1, we used $k = 5, \ldots, 10$, and $k = 6, \ldots, 10$ for dataset 2. For each kernel, the best regularization parameter for SVM was found using a gridsearch. One submission used Majority Voting and the other one used Weighted Majority Voting:

Model	Training	Validation	Private	Public
MV	0.97208	0.69833	0.66066	0.66933
WMV	0.99437	0.71417	0.66066	0.66400

Table 1: Scores of our chosen two submissions.

Our final result placed us in the **26-th place** of the leaderboard, with only two places being lost from the best ranking we had. While we feared for overfitting, it was not as drastic. We believe that we could have improved our results if we had used weaker SVM classifiers, since they would have allowed for more flexibility on the boosting model.

4.5. Further improvements

We know that a sum of kernels is a kernel and C times a kernel (with $C \geq 0$ is still a kernel. Therefore, instead of considering majority voting models, we could try to perform SVM using linear combination (with positive weights) of our best kernels and try to tune these weights to get the best possible performance. We also could have tried to use a mismatch kernel with higher values of mismatch m but we found the computation heavy even though our implementation was not that long.

	Dataset 0			Dataset 1			Dataset 2		
mismatch m = 1, k	Training Score	Validation Score	Best C	Training Score	Validation Score	Best C	Training Score	Validation Score	Best C
5.0	0.71875	0.62	0.0004	0.73	0.6075	0.0003	0.779375	0.7275	0.0002
6.0	0.7625	0.67	0.0002	0.833125	0.655	0.0004	0.8925	0.7225	0.0005
7.0	0.755	0.65	0.0001	0.865	0.6725	0.0002	0.954375	0.7575	0.0005
8.0	0.9775	0.67	0.0004	0.995	0.6725	0.0005	0.98125	0.7425	0.0004
9.0	0.983125	0.6675	0.0003	0.99875	0.66	0.0004	0.99625	0.765	0.0004
10.0	0.995625	0.6625	0.0006	1.0	0.685	0.0003	0.999375	0.7675	0.001
spectrum k									
5.0	0.72	0.65	0.002	0.791875	0.64	0.005	0.836875	0.7075	0.006
6.0	0.791875	0.6425	0.003	0.86625	0.6375	0.004	0.926875	0.7125	0.008
7.0	0.968125	0.6725	0.009	0.97375	0.6625	0.007	0.98375	0.705	0.01
8.0	0.9875	0.6275	0.01	0.998125	0.6575	0.009	1.0	0.7225	0.08
9.0	0.9975	0.605	0.02	1.0	0.6175	0.01	0.99875	0.7175	0.02
10.0	0.99875	0.5925	0.02	1.0	0.585	0.02	0.99875	0.66	0.02

Table 2: Best regularization parameters for SVM for each kernel. Found using a Grid Search.