Project Report

Karl Hajjar, Léonard Hussenot-Desenonges and Charles Maussion

"Sales Vauriens Machiavéliques"

1 Structure of the code

Our code contains four main files. The file **kernels.py** which contains all the functions used to compute different kernels (Polynomial, Gaussian, Spectrum and Mismatch). The file **learning_methods.py** contains two classes implementing the Kernel Ridge Regression (KRR) and Support Vector Machine (SVM) algorithms using a Kernel method (and thus taking as argument for training only a kernel matrix K and a vector of labels y). The file **tools.py** contains subsidiary functions useful for our training, testing and submitting pipeline. Finally the file **main.py** wrap-up all those functions to create our submission in the file Yte.csv. All the code is also available at `https://github.com/charlesmaussion/kernel`.

2 First tests

2.1 Kernel Ridge Regression

In order to familiarize with the methods and to quickly have our first results, we first worked on the numerical embedding given in the $X\_tr\_mat50.csv$ files. We first coded the **Kernel Ridge Regression**, which could be computed in **close form** and thus needed no external optimization machinery.

2.2 Linear, Polynomial and Gaussian Kernel

We first tested our KRR with the **linear**, the **2nd degree polynomial** and the **gaussian kernel**. This last one gave us a **0.65 score**, once the gaussian variance parameter was optimized. We did not further investigate other kernels as we preferred to start to work directly on the original DNA data.

3 Further Experimentations

3.1 Kernel on the the raw DNA data

Our first intuition was initially to use the **weighted-degree-kernel** but we then realized that we were not sure about the alignment of the different sequences and we would thus have to check for common sub-sequences, even localized at different places in the sequence. Investing the literature, we found that **spectrum** and **mismatch kernel** were two good candidates for our tests.
As we were working on raw DNA, which has a length 4 alphabet (A,T,C,G) and had no need for a very general code, we did not implement the most efficient algorithm for computing the mismatch kernel, which uses a suffix-tree structure and is sure useful when working on the length-21 protein alphabet but not that needed when working on raw DNA.
In order to build the mismatch kernel (the spectrum kernel implementation is almost the same as it is a 0-mismatch kernel and gave lower results), we built embeddings of each sequence in the form of a vector and a dictionary. For example, in a 1-mismtach kernel for subsequences of length 2, 'ACT' is represented by the vector [1, $\frac{1}{2}$, 1, $\frac{1}{2}$, $\frac{1}{2}$, 1, $\frac{1}{2}$, 1, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$]and the dictionary giving the sub-sequence/index correspondence : {'AC': 0, 'TC': 1, 'CC': 2, 'GC': 3, 'AA': 4, 'AT': 5, 'AG': 6, 'CT': 7, 'TT': 8, 'GT': 9, 'CA': 10, 'CG': 11}. The computation of the kernel then takes the form of a linear product between the embeddings.
After cross validation, we finally used the 11-2 mismatch kernel (subsequences of length 11 with 2 possible mismatches allowed) for the first two datasets, and the 8-2 mismatch kernel for the last one. We implemented kernel normalization : $K_{normalized} = K * k.k^T$ with $k = 1/\sqrt{diagonal(K)}$, but when cross-validating, we finally found at it was not giving us any more accuracy when predicting.

3.2 Support-Vector-Machine

We coded our SVM solving our optimization problem with *cvxopt*, a classical python optimization library. We expressed the optimization problem through its dual form as the expression of the constraints were easier to code in that form, and we even tried two different (yet equivalent) implementations of the dual problem to make sure our implementation was correct (both methods should yield the same results).
Trying to enhance the regularization power of our algorithm, we also implemented the 2-SVM, using the squared-hinge-loss, that showed even better performance, reaching **0.803** in accuracy over the public part of the test set, when optimizing the **penalization parameter** λ.

4 More things we could have tried

- Translate DNA into proteins (not that easy as we need to know where sequences begin) to work on proteins rather than raw-DNA : taking advantage of a priori human knowledge of the genetic code, its redundancy etc...
- Cross validate the choice of the penalizations for the 2-mismatch kernels. We have currently used a penalty of 1/2 for one mismatch and 1/4 for two mismatches, and did not investigate other values of the penalization as we would have had to recompute many times the mismatch kernels which is very time consuming but would probably have raised our score above 0.8.