Data challenge final report

Scores: *private* 0.242, 0.250, 0.320, 0.328
public 0.230, 0.254, 0.325, 0.332.

1 Introduction

Kernel methods are one of the solutions in the problems of machine learning area. In this project, we aim to solve a classification problem. It includes the vector images which are classified by 10 labels. We aim to find the best kernel method which can predict the class labels with high precision.

2 Methodology

In this section, we explain the kernel methods which are applied and strategies to train the model. We have used the *kernel ridge regression* with *Gaussian kernel (RBF)* for the classification purpose. However, we have written codes for kernel logistic and kernel support vector machine, but due to the fact that using these methods is time consuming, we did the final classification based on the kernel ridge regression. On the other hand, since we are dealing with a multi-class classification problem, we have used the *one-vs-rest* technique for the final purpose. For selecting the best model, we have used *cross validation* with 10 folds. Finally, for improving the scores, we have used the histogram of pixels rather than the raw data themselves. In the following sections we will discuss more about these techniques.

3 Multi-label Classification

For classifying a data set with more than two classes, we have two main techniques:
One-vs-Rest (OvR); which is a heuristic method and is used in the case of binary classification algorithms for multi-class classification. It is a proper strategy for raising up the accuracy of the model during the training part. More precisely, in this method we first set the labels of the data in class 0 to 1 and the rest to -1, and then obtain a classifier, say f_0, for this binary-type classification problem. We repeat this process each time by setting the labels of class i to 1 and the rest to -1. At the end of the day, we get 10 classifiers, say $f_0, \dots, f_9$. Finally, we set the label of an unseen data d equals $\arg\max_i f_i(d)$.
One-vs-One (OvO); again is another technique in multi-class classification problems. In this method, for each pair of class labels i and j, we train a classifier f_{ij} on the data with labels i and j, by setting the labels of the data in class i to 1 and the data in class j to -1. Now for an unseen data d, we set the label equals $\arg\max_i \#\{j | f_{ij}(d) > 0\}$.
We have applied both these strategies for the data challenge. We gain the same results on both methods. We also applied a mixture of these two, by first using the OvR method to obtain the two most probable labels and then using the OvO to decide which one to choose as the final label. Unfortunately, it did not make any notable change in the final scores. In the notebook file, we have provided the code only for OvR.
Why we preferred OvR over OvO? Firstly, they have both the same results. Secondly, for each classifier we have two parameters, namely, σ and λ. Hence using OvR, left us 20 parameters while OvO requires 90 parameters! However, both of these two numbers are large, yet, 20 is preferable.

4 Kernel Tricks

In what follows, we will discuss about different kernel trick methods we have used for classification and their advantages and disadvantages. It is very important to clarify that for us, the most important thing for choosing one method over another was first *time* and then *accuracy*. Because as it was mentioned in the previous section, we have 20 parameters for training each model, and hence, being able to modify this large number of parameters quickly and observe the results shortly after, is only possible if we have written an optimized code.

4.1 Kernel Ridge Regression

As the first kernel, we applied kernel ridge regression. It is the most elementary kernel, yet the fastest one, that can be applied. We remind that the coefficients α is obtained by

$$\alpha = (K + \lambda n I)^{-1} y$$

It is very important to clarify that in the code, we have not obtained α by calculating the above inverse matrix. This would be time consuming. Instead, we have used the function `numpy.linalg.solve` which solves the equation $(K + \lambda n I)\alpha = y$ in a couple of seconds.
Improvements: At the beginning, we computed the gram matrix K using two nested `for` loops, each of which iterated for 5000 times! This took 5 minutes to complete. Later, we used the function `euclidean_distances` from the `sklearn.metrics` package. Given two matrices X and $\tilde{X}$, it returns back a matrix M where its M_{ij} element is the

ℓ^2 distance between the i^{th} row of X and j^{th} row of $\tilde{X}$. The advantage of this method is that it computes this matrix in 1-2 seconds. Finally, by applying `numpy.exp` on this matrix we obtain the gram matrix in only 2-3 seconds which is a win!

4.2 Kernel Logistic Regression

Kernel logistic regression can be expressed in terms of an iterative method which has re-weighted part in each iteration. We used exactly the same algorithm taught in the course. More precisely, we start by setting $\alpha^0 \leftarrow \mathbf{0}$, updating the weight W^t and z^t at time step t using α^t, and then updating α^t by

$$\alpha^{t+1} \leftarrow \texttt{solveWKRR}(K, W^t, z^t)$$

where K is the gram matrix and `solveWKRR` minimizes the standard *weighted kernel ridge regression* problem

$$J_q(\alpha) := \frac{1}{n}(K\alpha - z^t)^\top W^t (K\alpha - z^t) + \lambda \alpha^\top K\alpha + \text{constant}.$$

Since Newton method has been used in this algorithm, it converges very fast. We set the stopping criteria to error $:= \|\alpha^{(t+1)} - \alpha^t\|_2 < 0.0000001$. Usually, after 5-10 iterations this criteria is satisfied. In practice, it takes 10-15 seconds to complete which is nice, yet, still slow compared with KRR which takes only 2 seconds!
Improvements: At the beginning, we treated the weights W as diagonal matrices and used `numpy.matmul` for multiplying these matrices. Later, we treated them as simple 1d arrays and used the operator $*$ for multiplication purposes. It decreased the elapsed time from 2m 15s to just 15s!

4.3 Kernel Support Vector Machine

In Kernel Support Vector Machine, we face a quadratic problems as follows:

$$\max_\alpha \ \sum_i \alpha_i y_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j K(X_i, X_j) = \alpha^\top y - \frac{1}{2}\alpha^\top K\alpha$$
$$s.t \ \ 0 \le \alpha_i y_i \le \frac{1}{2n\lambda} = C \qquad i = 1, \dots, n$$

This is the dual problem of SVM. To find the solution of this quadratic problem with constraint, we applied the `cvxopt` package. It uses an algorithm which converges quickly compared with other methods. The maximum number of iterations in this package is 10 by default which takes about 1m 30s - 3m 30s for solving one single problem with the same dimension as ours. The disadvantage of this method is that for training 10 classifiers and using 10-folds cross validation, it takes 2h 30m - 6h for obtaining the results. Now, imagine that we change the parameters σ and λ. It would be a nightmare!
Hence, we decided to find a faster method. We tried to find some packages with the ability to parallelize solving these problems, but, it was all in vain. We also tried to do this by concatenating 10 α's and convert it into a single problem. This time, we faced the error *"out of memory"*!

Finally, we decided to go to the *coordinate ascent* algorithm explained in exercise 16. The following table compares this algorithm with the `cvxopt`,

		$\lambda = 0.1$	$\lambda = 0.01$	$\lambda = 0.001$	$\lambda = 0.0001$
cvxopt	*maxiters*	10	10	10	10
	elapsed time	2m 03s	2m 02s	1m 46s	1m 35s
	$\alpha^\top K\alpha - 2\alpha^\top y$	-5.28447164	-27.168826723	-230.162584227	-1891.9076066
co asc	*maxiters*	18	183	3103	147000
	elapsed time	1s	8s	2m 17s	1h 37m 05s
	$\alpha^\top K\alpha - 2\alpha^\top y$	-5.28447175	-27.168826741	-230.162584233	-1891.9076078

In the above table, the *maxiters* for the coordinate ascent method is the number of iterations required to achieve an accuracy better than the `cvxopt`. Here, by an *iteration* for coordinate ascent we mean a complete update of the whole coordinates of α. As you may see, for $\lambda < 0.01$ `cvxopt` is the winner while for $\lambda > 0.01$ using coordinate ascent is cheaper! We guess that the reason for this is that as λ gets larger, C gets smaller, and hence, the convergence of α_i's will be faster, because they live in the interval $[-C, C]$.
In practice, we need small λ's ($< .0001$). We used SVM without cross validation for these amount of λ's by applying `cvxopt`. The accuracy was higher on a test set compared with other kernel tricks, however, because of the matter of time we decided to continue with KRR.

5 Histogram Trick

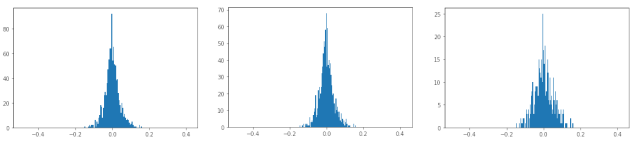
We apply the histogram trick which is a helpful method to normalize our data set. The process begins with mapping the raw data $[RGB]$ to $[\varphi(R)\,\varphi(G)\,\varphi(B)]$ where for each $X \in \mathbb{R}^{1024}$

$$\varphi(X) = \texttt{numpy.histogram(X, bins=b, density=True).}$$

The vector `b` determines the partition of the interval $[-0.49, 0.424]$ over which the histogram is calculated. This interval is the range of pixel values.
First, we partitioned this interval into sub-intervals with the same size. Doing so, one obtains histograms which are populated near the origin and sparse elsewhere, figure (a)! Later, we partitioned this interval into $[x_{-n}, \dots, x_n]$ where

$$x_i := c\,\frac{i}{2n}\,e^{\frac{(i/2n)^2}{2s}}$$

for given n and s. Here, c is a scaling factor. The advantage of histograms obtained through this method is that we can have more control over them. More precisely, the foregoing partition has the property that as we get closer to the origin, the length of each sub-interval gets smaller. In fact, by decreasing s, we can expect more uniform histograms; $s = 0.5$ and $s = 0.08$ in figures (b) and (c) respectively.



(a) (b) (c)

This method gave higher scores in practice.