UNIVERSITÀ DEGLI STUDI DI MILANO

Academic year 2023/2024

Experimental project of Statistical Methods for Machine Learning

Project 1: Kernelized Linear Classification

Antoine Guines

Contents

1 Organization of the dataset **5**

2 Pre-processing techniques **5**

3 Algorithms implemented **6**

3.1 Perceptron algorithm 6

3.2 Pegasos algorithm for SVM 6

3.3 Pegasos algorithm for Regularized logistic classification 7

3.4 Kernelized Perceptron with polynomial and gaussian kernel 8

3.5 Kernelized Pegasos for SVM with polynomial and gaussian kernel 8

4 Hyperparameter tuning **9**

5 Comments and discussion on the experimentals results **10**

Introduction

This report presents the work carried out as part of the final experimental project for the Statistical methods for machine learning course. The project I chose is the project number 1 on linear classification using algorithms such as the Perceptron, SVM, and their kernelized version. This report is accompanied by a notebook containing the code of this project, organised into severals parts, each part corresponding to the implementation of one specific algorithm. This report will be divided into several part. First we will talk about the organization of the dataset, and the way we have pre-processed the data. Then we will go through the algorithm we have implemented, before describing the way we choose the hyperparameter. Finally we will have some discussion on the experimental results.

1 Organization of the dataset

The dataset we were working on was initially a .csv file with 10,000 rows and 11 columns. Each row corresponded to a point in the dataset, with the first 10 columns associated with a 10-feature X vector and the last column corresponding to the label associated with this vector. The 10 features of the X vector were represented as real numbers, and the Y label as -1 or 1. This dataset can therefore be used for binary classification, since each X vector is associated with a label $Y \in \{-1, 1\}$. We will now describe how we have pre-processed this dataset.

2 Pre-processing techniques

In this section, we will detail the pre-processing steps that we applied to the dataset, in order to be able to process it easily after.

First, we load the dataset from a .csv file into a pandas DataFrame, enabling inspection of its structure and content by examining its dimensions and how looks a few rows. As mentionned in the previous part this dataset has 10000 rows and 11 columns.

Subsequently, then we check for missing values in the dataset. This process is very important if there are some missing data since it can then affect model performance. But in our case, no missing value were found.

After the missing value check, we first shuffle our dataset and then we normalizes it, in order to scale all the features between 0 and 1. This is design to improve the performance and stability of the algorithms we will use by ensuring that no single feature dominates the learning process due to its scale.

Then once our data are normalized, there's another important step to take. Indeed we must not forget to add one more component to our feature vector X, and this component must be equal to 1. Throughout this project, we will be working on linear classification algorithms, where the predictors are of the form $w^T x + b$. By adding a component to x (a 1) and a component to w (the b) we can return to the homogeneous case, which corresponds to a classifier of the type $w'^T x'$. This will simplify our task for the rest of the work, because by adding this component we return to the homogeneous case, which is simpler to process.

Finally, we split the dataset into training and test sets. The training set is composed of 80% of the data, that is to say 8000 examples, while the remaining 20% are for the set, that is to say 2000 examples. This separation of the dataset into a training set and a test set from the start ensures a good separation of the data and that there should be no leakage from one to the other. In this way, the data in the test set used to evaluate the algorithms will never have been seen before by the algorithm, so there will be no bias in the results on this side. Now that our data is ready, it's time to move on to implementing the algorithms.

3 Algorithms implemented

In this project we implemented 5 different algorithms. In order we implemented the Perceptron algorithm, Pegasos algorithm for SVM, Pegasos algorithm for Regularized logistic classification, Kernelized Perceptron with polynomial and gaussian kernel, and Kernelized Pegasos also with polynomial and gaussian kernel. Let's describe them the one after the other.

3.1 Perceptron algorithm

Here is my code for the perceptron algorithm.

Algorithm 1 Perceptron

```
self.w = np.zeros(self.feature_number)
self.max_epoch = param

for epoch in range(self.max_epoch):

        update = False
        for i in range(X_train.shape[0]):

            dot_prod = Y_train[i] * np.dot(self.w, X_train[i,:])

        if (dot_prod <= 0):
                self.w = self.w + Y_train[i] * X_train[i,:]
            update = True

        if (update == False):
            break
return self.w
```

For each epoch, we go through all the examples in the training set. For each example, the algorithm makes a prediction. If the prediction is wrong, we update the linear classifier vector. If the prediction is correct, we do nothing.

Just like in the theoretical case, we keep the update parameter, which allows the loop to exit early if all the predictions are correct. However, it is very unlikely that this will happen because the dataset is probably not linearly separable.

3.2 Pegasos algorithm for SVM

Now let's see my code for the Pegasos algorithm for SVM.

Algorithm 2 Pegasos for SVM

```
self.T = T                               # T is the number of rounds
self.lambda_param = lambda_param         # lambda is the regularization coefficient

sum_h = 0
h = np.zeros(self.feature_number)
sample_size = X_train.shape[0]

for t in range(self.T):

    lr = (1 / (self.lambda_param * (t + 1)))
    i = np.random.randint(0, sample_size)
    condition = 1 - (Y_train[i] * np.dot(h, X_train[i,:]))

    if (condition > 0):
        grad_l = self.lambda_param * h + Y_train[i] * X_train[i,:]
    else:
        grad_l = self.lambda_param * h

    h = h - lr * grad_l
    sum_h = sum_h + h

self.w = (sum_h / self.T)

return self.w
```

For each round, we calculate the learning rate and we randomly select an example from the training set. Then we calculate the condition from the Hinge loss in order to know the expression of the gradient we will use. If the condition is greater than zero, we update the gradient with both the regularization term and the selected example. Otherwise, we update the gradient with only the regularization term.

We then do the usual step of gradient descent, and at the end of the algorithm we calculate the linear predictor as the average of all previous predictor over all rounds.

3.3 Pegasos algorithm for Regularized logistic classification

This algorithm is almost the same than the previous one since the only difference is the loss function use for the objective function of Pegasos. In the SVM case it was the Hinge loss, while now we use the logistic loss. Therefore the only difference resides in the way we compute the gradient descent. In this case the expression of the gradient will be compute as :

Algorithm 3 Pegasos for Logistic classification

```
        num = Y_train[i] * X_train[i,:]
        den = np.log(2) * (1 + np.exp(Y_train[i] * np.dot(h, X_train[i,:])))

        grad_l = self.lambda_param * h - (num / den)
```

The others steps stay unchanged.

3.4 Kernelized Perceptron with polynomial and gaussian kernel

This code is the version of the Kernelized Perceptron algorithms.

Algorithm 4 Kernelized Perceptron

```
K = self.kernel(X_train, X_train, self.param_kernel)

for epoch in range(self.max_epoch):

    update = False
    for i in range(sample_size):

        pred = np.sum(self.w_index * Y_train * K[:, i])

        if (np.sign(pred) != Y_train[i]):
            self.w_index[i] += 1
            update = True

    if (update == False):
        print('no update in last epoch')
        break
```

The operation is similar to that of the classic perceptron, with the key difference being that the feature vectors X lives in a higher-dimensional space where their product is computed using a kernel function. Otherwise, the process remains the same: When a prediction is incorrect, the weight vector w (now representing the count of misclassifications for each example) is updated, otherwise nothing.

3.5 Kernelized Pegasos for SVM with polynomial and gaussian kernel

Finally here is the code of the last algorithm implement in this project.

Algorithm 5 Kernelized Pegasos for SVM

```
K = self.kernel(X_train, X_train, self.param_kernel)

for t in range(self.T):

    lr = (1 / (self.lambda_param * (t + 1)))
    i = np.random.randint(0, sample_size)

    condition = Y_train[i] * lr * np.sum(self.w_index * Y_train * K[:, i])

    if (condition < 1):
        self.w_index[i] += 1
```

The code works in the same way than the one of kernelized perceptron, but with a different condition. As mentionned in the project subject, the pseudo code was given in this paper [1], and there was an error in the update condition. In the paper, the pseudocode instructed to use Y_train[i] in the sum, whereas we should use all Y_train[s] where s are all the indices associated with non-zero weights in w. Therefore, in our case, we simply need to use Y_train, and multiplying it with w we will keep only the indices associated with non zero weight.

4 Hyperparameter tuning

Now, let's see how we chose our hyperparameters for the different models. To select these hyperparameters, we started by dividing our training set into two parts: a smaller training set (X_train_tuning, Y_train_tuning) and a validation set (X_val_tuning, Y_val_tuning). Then, following different algorithms, various parameters have to be chosen, but overall, we consistently apply the same methodology.

For example for the perceptron algorithm, the only hyperparameter we need to choose is the number of epochs. We create a list containing different epoch values, ranging from 1 to 30. Among these values, we aim to find the best one. We input this list into our tuning_parameter function. This function then trains the algorithm for each epoch value in the list using X_train_tuning. Subsequently, it evaluates the predictor on X_val_tuning and stores the result (the accuracy) in an array. This process is repeated for each epoch value. Once completed, we identify the maximum accuracy from the array containing all results and retain the epoch value that yields this result. This selected hyperparameter is then used to retrain the algorithm on the entire training set X_train, followed by evaluating the predictor on the test set X_test.

Then for SVM we have to choose T, the number of rounds, and λ the regularization coefficient. This time we have to select 2 hyperparameters, but finally it changes nothing, we just create 2 list with different value of each hyperparameters, and then we test all possible combinations. In the end, we select the combination that yields the best result. Of course, it's essential not to provide too many choices for each hyperparameter to ensure it remains feasible within a reasonable time frame.

We repeat the same process for the different hyperparameters of all our algorithm.

5 Comments and discussion on the experimentals results

Firstly, let's specify the experiments conducted during this project. As mentioned earlier, we ran five algorithms: the Perceptron algorithm, Pegasos algorithm for SVM, Pegasos algorithm for Regularized Logistic Classification, Kernelized Perceptron with polynomial and Gaussian kernels, and Kernelized Pegasos also with polynomial and Gaussian kernels. In addition to these five algorithms, we also applied the Perceptron, Pegasos for SVM, and Pegasos for Logistic Linear Classification with expanded features of degree 2. For this purpose, we had to create a function that takes our feature vector X, consisting of 11 features, and applies a transformation to obtain a vector with 66 features corresponding to the degree-2 expansion of X. Next, we were able to use the same algorithms as before, specifying that the size of the predictor should also be 66, as we were looking for a predictor belonging to a higher-dimensional space. This predictor, having more information, is expected to yield better results than if we had remained in a space with fewer features.

Now let's move on to the results. The metric chosen to evaluate algorithm performance is logically accuracy, which simply counts the number of correct predictions out of the total number. It is thus the percentage of correct answers from our predictor.

For readability, we will present the results in the following table:

Table 1: Results

Algorithms	Hyperparameters	Accuracy on test set
Perceptron	epoch = 1	0.73
Perceptron exp. feat. of degree 2	epoch = 27	0.89
Pegasos for SVM	T =10000, $\lambda = 0.01$	0.5
Pegasos for SVM exp. feat. of degree 2	T =1000, $\lambda = 0.01$	0.5
Pegasos for logistic classification (LC)	T =50000, $\lambda = 0.001$	0.73
Pegasos for LC exp. feat. of degree 2	T =50000, $\lambda = 0.001$	0.75
Kernelized Perceptron pol. kernel	epoch = 41, degree=14	0.92
Kernelized Perceptron gaus. kernel	epoch = 41, $\gamma = 0.1$	0.94
Kernelized Pegasos SVM pol. kernel	T = 30000, $\lambda = 0.1$, deg=14	0.87
Kernelized Pegasos SVM gaus. kernel	T = 30000, $\lambda = 0.001$, $\gamma = 0.1$	0.86

Overall, the different algorithms achieve quite good results, all well above random guessing, except for the SVM which has an accuracy of 0.5, equivalent to random prediction. The perceptron is the one that seems to achieve the better result. And as expected, combining features and thereby enabling learning in higher-dimensional spaces with kernels significantly improves results. Specifically, the Kernel SVM performs very well this time, although inferior to the kernelized perceptron which has the highest accuracy, achieving nearly 95% accuracy.

Conclusion

To conclude, this project ends with results that I find quite satisfactory. We have observed how using kernels and combining features significantly improves learning in the context of linear classification.

Bibliography

[1] : https://home.ttic.edu/~nati/Publications/PegasosMPB.pdf