MPML report

Filippo Calzavara, Nicolae Righeriu

December 2018

1 Introduction

The project consists of creating a parallelized version of Logistic Regression using Python and Spark libraries to solve a classification problem. The dataset used for this assignment is the Spam dataset, downloaded from `https://web.stanford.edu/~hastie/ElemStatLearn/data.html` under "spam.data". Parallelism is important when dealing with large data sets, however it raises challenges when performing computations since there is no guarantee they are done sequentially. The parallel concepts of map-reduce and filtering from the Spark library are frequently used throughout the code and with their help, the computations of Logistic regression benefit the a speed up of up to 2 times faster than the regular single threaded approach.

2 Data set description

The dataset consists of 4601 data points, each containing 58 attributes out of which 57 are continuous and 1 is a nominal class label (0 not spam and 1 is spam). There are 1813 spam instances, corresponding to 39,4% of the data set. Mostly the attributes indicate whether a particular word was frequently appearing in the e-mail with a score from 0 to 1. Attributes 55 to 57 measure the length of sequences of capital letters. For a complete in-depth description of all the attributes in the data set, visit the link written in the introduction and consult the spam.info file.

3 Algorithm implementation

3.1 Data Preprocessing

Before conducting any Machine Learning, a preprocessing phase of the dataset has occurred, that firstly consists in removing column 57, as indicated in the hints.
As second step, the data are collected into a column wise structure (number of features times number of samples) which will later be useful for data normalization. Then each tuple is being keyed and depending on the constant TRAIN_TEST, as 1 if it should belong to the train set or 0 if it should belong to the test set. Then the data is normalized as described in the subsection below and a new tuple is returned from the map preprocessing containing the random key, the array of values for each features, the structure containing the value of feature, and the classification target. Afterwards the tuples are sorted by key. This can be seen as shuffling the data, since the key is assigned in a random way.

3.2 Data Normalization

Given the structure before mentioned, a function to calculate the average and the variance of the value for each column is called.
Then, the data is normalized by calculating the average of each row and subtracting it from each value and dividing by the corresponding sigmas of each corresponding row. The sigmas are calculated with the formula:$s_{x_i} = \sqrt{\frac{\sum(x_i - \overline{x})^2}{n-1}}$, where $\overline{x}$ is the average of the column. To store the averages, sums and sigmas for each row accumulator data structures specific to Spark are used.

3.3 Training

In the beginning, the training function is called for cross validation (CV) purposes, to find the best hyperparameters which are learning rate and lambda regularization. Firstly the accumulators and weights are initiated, the weight and bias starting out with values 0. Then the data is split with a ratio of 80% training and 20% test. In the cross validation described below the data is further split into k-Folds. Afterwards the best hyperparameters found are applied to the whole training set and the weights are "trained". In the finalization these trained weights are tested on the test set.
Gradient descent is ran in both the k-Fold CV and, in the final computation of the errors with the best hyperparameters found in CV on the whole training set. It will run the number of iterations which it receives as a parameter and compute the cost for each iteration, which is calculated using the same formula as in the project description.
$J(W) = -\frac{1}{m}\sum_{i=1}^{m}(y^i log(\hat{y}) + (1-y^i)log(1-\hat{y}^i)) + \frac{\lambda}{2m}\sum_{i=1}^{k} w_i^2$, where the predictions are calculated using the predict() function in the code. The compute_cost() function calculates in its final call the precision, recall and f1-score based on the hits and misses. To enable monitoring the process, the cost of gradient descent is printed in every iteration to the console. To classify each example as spam or not, the probability is rounded up or down depending if the value is higher or lower than the threshold set (here default is 0.5).
In each call of the gradientDescentIteration() function the derivatives are computed parallel using the spark map() and sum() functions. Then the weights and bias are adjusted depending on the learning rate.

3.4 Paralellization

Paralellization is employed in various places throughout the code, there are however some restrictions when it comes to performing sequential operations in parallel and therefore are done single-threaded. Specifically the paralellized operations are:

- preprocessing and labelling data as train/test which helps later in splitting it
- shuffling the data and calculating the residuals for normalization
- splitting the data to be used for k-Fold CV
- predicting spam/not spam label and computing weights in gradient descent
- filtering predictions to compute confusion matrix for accuracy measures
- computing gradient descent cost

These operations are done using spark functions like map(), filter(), sortBy() (the samples are sorted by the random assigned keys to them and are therefore shuffled), zipWithIndex() (for easing the split for k-Fold) and count(). As a requirement of the project, no external libraries such as numpy or pandas are used and instead the datasets are converted into a Spark formats with sc.parallelize() and accumulators are used for all the computations which involve loading and performing computations on the dataset.
The operations done within k-Fold CV are done in parallel, however each of the k-Folds is calculated sequentially as it is not possible to perform parallel operations within something which is already running in parallel.

4 Cross validation procedure

To evaluate the goodness of the hyperparameters (learning rate and lambda regularization), the kFoldsCV function splits the data into k parts, where k is a inputted parameter. Of these, k-1 parts serve for training and one is left out for testing. The part chosen to serve for testing is done in a windowed way, like a window of size $\frac{trainingsetsize}{k}$ moving across the whole training set. To do this the data is indexed using the zipWithIndex() spark function and after filtered depends on its index. Then gradient descent is applied for the training set while all the costs are saved and printed to show progress. Finally, in each fold the trained weights are tested and the errors and computed on the held out test set (which here is still actually part of the big train set).

To find the best hyperparameters a grid is created with a range of learning rates and lambdas. For each of these pairs k-fold CV is ran to get a reliable evaluation of them.

5 Experiments

Firstly the k-Fold CV is performed. For this, a grid was created containing different combinations of hyper-parameters, with learning ranging from 0.09 to 0.54 and lambda regularization from 0.028 to 0.364 and 4 values in between the interval for learning rate and 2 for lambda. The best parameters found were 0.36 for learning rate and 0.196 for lambda. The iteration number was set to 50, as the descent stopped after this amount, as can be seen in figure 1.

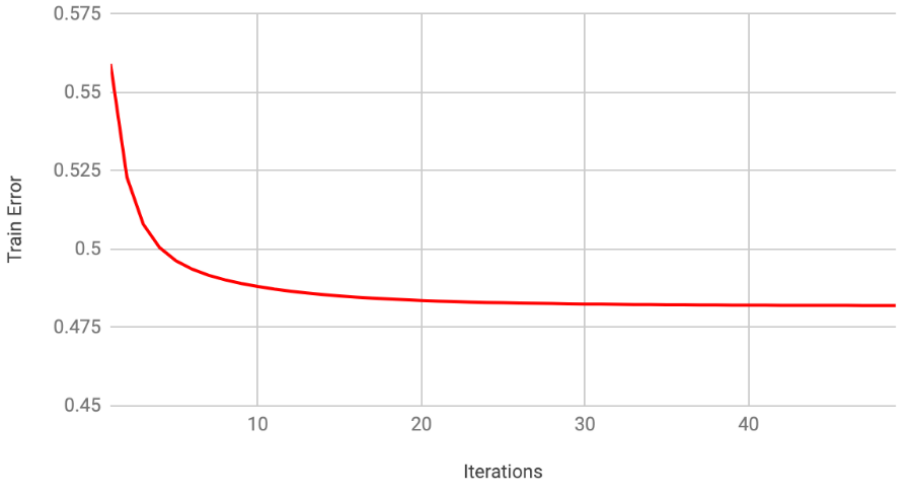

Figure 1: Showing train error cost stagnating over 40

Depending on the number of workers set, the speedup can increase or decrease. As the machine on which the experiments were run on has 4 cores, it can be observed in figure 2 that the speedup plateaus after 4 workers and even decreases after 8, since the machine has 8 threads.

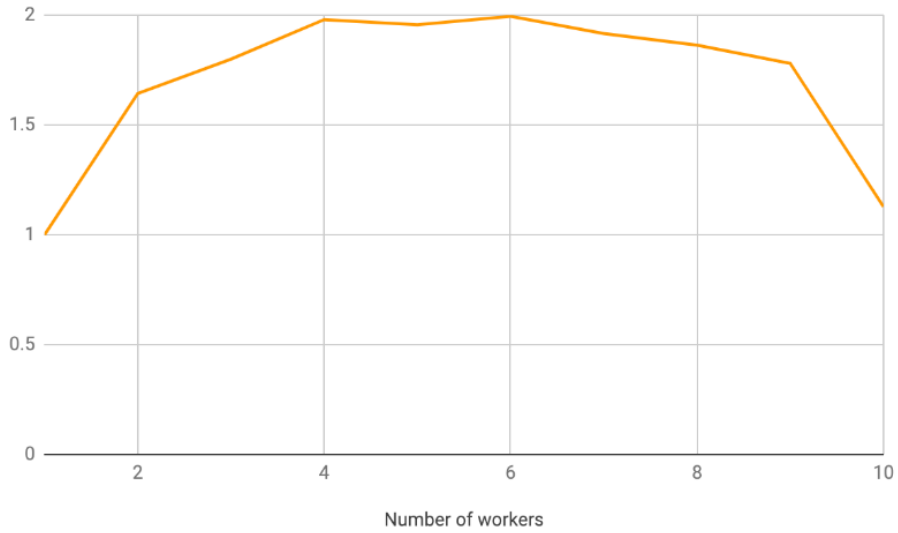

Figure 2: Speedup, showing ratio of $\frac{\text{running time with 1 worker}}{\text{running time with n workers}}$

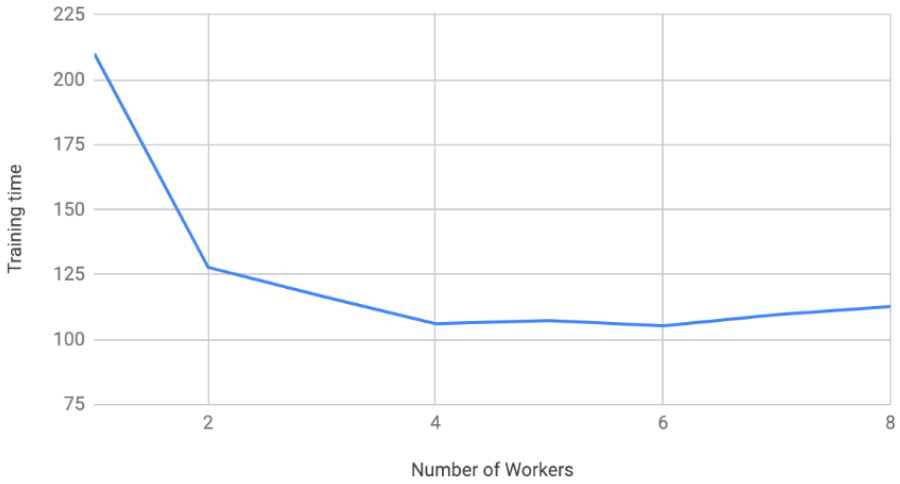

Figure 3: Showing training time, with a varying number of workers

Regarding performance, in figure 4 it can be observed the percent of classifying false positive is very low. This is highly desirable since classifying an email as spam when it is not should be avoided more than not classifying it as spam when it is. Failing achieve a small false positive classification percent can lead to somebody missing important emails as they are classified as spam. These rates can be adjusted by changing the threshold value (which is default 0.5). A higher value means that less emails will be classified as spam. For example a threshold of 0.7 will cause that also emails with a cost between 0.5 and 0.7 won't be classified as spam, besides the default ones below cost 0.5.
The result statistics of the best model are as follows:

- Precision: 94.25%
- Recall: 56.44%
- F1-Score: 70.61%
- Accuracy: 80.68%

True Positive	False Positive
23.20%	1.41%
False Negative	**True Negative**
17.90%	57.48%

Figure 4: Results on the test set with the best hyperparameters

6 Conclusions

The Machine Learning algorithm implemented allows parallelizing the pre-processing and gradient descent operations. The implementation of a grid allows retrieving the best hyperparameters in an automated fashion.

The strength of the implementation lies in a significant speedup curve of the running times of up to 2 times the running time of conventional single threaded approaches and relying solely on Spark data structures with minimal use of external libraries. Accuracy and precision are good and, because of the small dataset the model can be learned in a relatively small amount of time and iterations. Moreover, a log system was created which exports relevant data processing time and details.