Parallel Implementation and Evaluation of a supervised ML algorithm

Massively Parallel Machine Learning
(2018-19)

Project Report

Universidad Politécnica de Madrid

Date: 18.12.2018

Professor: Alberto Mozo
Students: Moritz Meister,
Marcin Paszkiewicz

Introduction

In the past few years a great variety of general-purpose cluster-computing frameworks emerged that allow for distributed, high-performing, fault-tolerant and accurate data analytics applications, such a framework is the by the Apache Foundation published open source project Apache Spark.

The aim of this project is to implement Logistic Regression in a parallelized fashion leveraging the capabilities of Apache Spark and subsequently evaluate the implementation in terms of prediction performance and speed-up for different numbers of worker machines.

The problem was approached in the following order: As a baseline, a standard Gradient Descent (GD) approach was implemented, which was further optimized by extending it for the possibility to use a Stochastic Gradient Descent (SGD) approach. Furthermore, a Cross-Validation (CV) procedure was implemented in order to be able to tune hyperparameters and reliably estimate the performance of the model in terms of generalization capability.

The report is structured as follows: Firstly, the dataset used for the project is described in detail. Secondly, details of the implementation decisions are explained, for example of the training and validation is parallelized. This is followed by a description of the implementation details for the CV and some findings regarding the tuning of hyperparameters. The fifth section contains the experimental results regarding performance and speed of the implementation. The report is concluded by some general remarks, difficulties and findings of the project.

Data Set Description

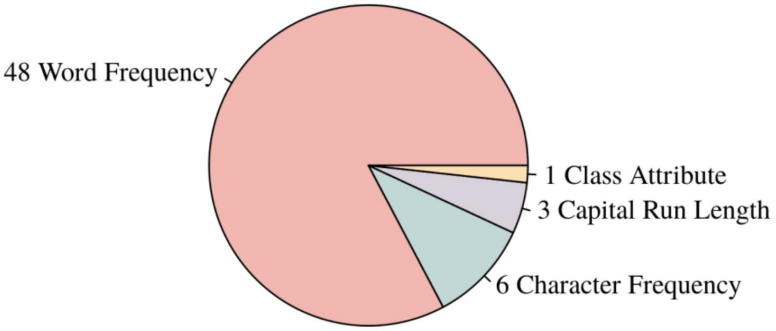
As a dataset for the evaluation of the project the well known “SPAM E-mail Database”[1] was used. The database was created in the Hewlett-Packard Labs (HP) for the purpose of classification of emails as spam or non-spam. False positives (marking good mail as spam) is very undesirable in this machine learning task. According to HP, if we insist on zero false positives in the training/testing set, 20-25% of the spam would pass through the filter.

The dataset contains 4,601 instances (each of them representing one email) of which 1,813 are classified as spam, hence we have some minor class imbalance, which shouldn’t influence our models too much.

Furthermore, the dataset contains 58 column, of which 57 are continuous features and 1 is the nominal class label. Most of the attributes indicate whether a particular word or character was frequently occuring in the email. The run-length attributes (55-57) measure the length of sequences of consecutive capital letters. However, the project requirements specify that the 57th column should not be used and is to be removed from the dataset. Probably for the

[1] https://web.stanford.edu/~hastie/ElemStatLearn/data.html

reason that it is on a completely different scale than the other variables. The dataset does not contain any missing values.



Looking at the distributions of the features, there are no surprises except for a few features that might contain outliers. The following table shows the distribution statistics of some of the features and the target (column 58):

column	min	max	mean	st. dev.
27	0	33.33	0.7673	3.3673
52	0	32.478	0.26907	0.81567
55	1	1102.5	5.1915	31.729
58	0	1	0.39404	0.4887

As an example, Columns 27 and 52 have a quite low mean, but their maximum values are much higher, indicating that they contain some outliers. Column 55 is on a different scale but also the maximum value is much higher than the mean value. However, since our approach contains regularization, it should penalize the features with less predictive power and decrease their weights in order to limit their influence on the model.

Algorithm Implementation

This section will explain the details of the algorithm implementation. We started implementing the basic gradient optimization procedure for the Logistic Regression problem and added stochastic gradient as an extension later.

The Learning Problem and Points of Parallelization

Loss function to be optimized:

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(\hat{y}^{(i)}) + (1-y^{(i)})log(1-\hat{y}^{(i)})) + \frac{\lambda}{2m}\sum_{i=1}^{k} w_i^2$$

2. Step — 1. Step

Derivatives:

$$dw_1 = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)}) * x_1^{(j)} + \frac{\lambda}{m} w_1$$

2. Step — 1. Step

In the logistic regression problem, we aim at learning a mapping from the feature space to to the categorical outcome (0 or 1). For this problem, it was shown that the cross-entropy loss is also the maximum likelihood estimator for the weights, and hence it can be used to learn good weights. The loss function to be optimized is shown above, accompanied with its derivative (shown for one weight only, others are derived in the same way).

Analysing the loss function and its derivative, one can see that there are two major operations that can be translated into the map and reduce logic of Spark.The operations marked with “1. Step” are row-wise vector operations, while the summation is over all observations. Hence, we can parallelize the 1. Step with a map function and the 2. Step will require a reduce step:

1. Step: map
2. Step: reduce

The map step will be executed in parallel where all workers will perform the map step on their chunk of data. The reduce step can first be executed on the worker but once the worker reduced its data chunk it will require communication to reduce across the workers.
This is the degree of parallelism one can reach during model training.

As you probably have noted in the figure above, there are some division by the number of observations m, which have been crossed out. We removed these divisions in our implementation, in order to have small values for the lambda, otherwise you need to try very large values and we are used to this variant with small lambdas. This is merely a change for convenience, but does not influence the model or convergence, only the lambda value itself does.

Furthermore, the calculation of predictions can be parallelized with a map step, since it is a row-wise operation and therefore all workers can perform it on their data partition independently.

Data Utils and Standardization

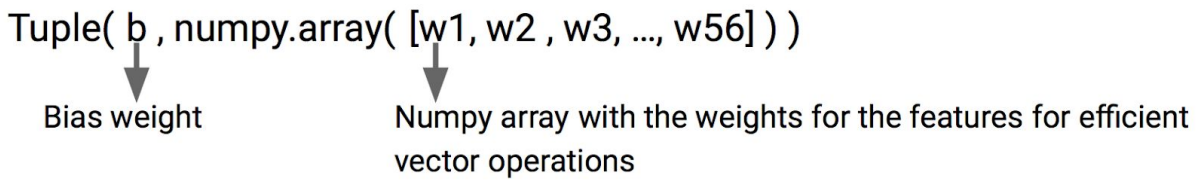
In order to handle reading and standardizing the original data, we decided to implement a *DataUtil* helper class, which takes as input the following parameters:

sc:	A SparkContext.
inputPath:	The file path to the input file
statsInputPath:	The file path to a file containing the mean and standard deviations of the features.
standardize:	A boolean indicating whether the data should be standardized to zero mean and unit variance.

The *DataUtil* provides a read method to read the file and return a RDD, if standardize is set to true it returns the standardized features. The read function also deletes column 57 as specified in the requirements.
The data is standardized with the means and standard deviations read from the file specified with the *statsInputPath*. We are not recomputing the means and stdevs. each time since in a real world scenario, where the amount of data is much bigger, you would want to avoid doing this costly operation multiple time, but instead do it once and safe the results. Furthermore, once you train a model with a certain standardization, you want to standardize future observations with the same statistics and only recompute them once you also retrain the model. That is because the model learns a certain distribution of the features and if another standardization is applied, it is likely to fail and lose performance.

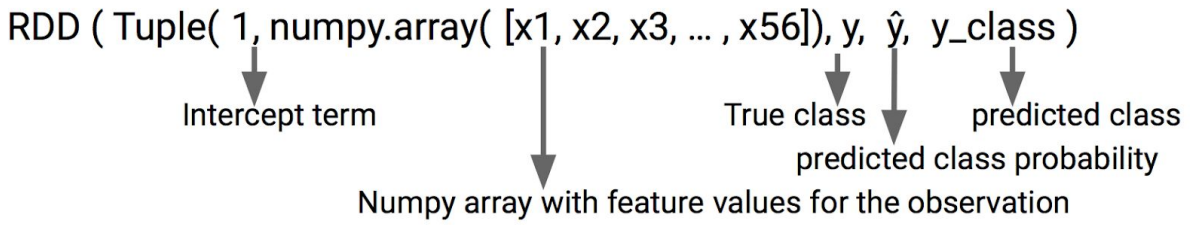
Data Structures

To parallelize operations in a convenient way and to be able to leverage vector operations for row-wise calculations we chose a certain data structure. One for the learned weights, since the bias term (the weight for the intercept) should not be regularized, we are keeping it separate. The weights are saved in a Tuple, where the first position contains the bias weight and at the second position contains a numpy array with a length equal to the number of features in the data set for the weights of these features. The following picture illustrates the tuple:



The other data structure serves to keep the data set, it is a RDD where each row is a tuple containing the information of one observation. The tuple contains a 1 for the intercept at the first position, the second position holds a numpy array with the rest of the feature values, the third position contains the true class value that is to be learned (0 or 1), the fourth position

holds the predicted probability of the logistic regression during and after the learning process and the last position will be filled with the predicted class once the training is finished, such that the error metrics like accuracy, precision and recall can be computed. The data structure is illustrated below:



Furthermore, the learned models, with all the parameters and error metrics are saved and returned in dictionaries.

Training

All the functionality to train, validate or cross-validate a model is contained in a Python class *ParallelLogReg.* The class is initialized with the following parameters:

data:	RDD as read by the DataUtil containing all the data.
Iterations:	number of iterations to be used in the Gradient Descent optimization.
learning_rate:	Learning rate for the GD.
Lambda_reg:	Regularization parameter for the L2-norm on the weights (except bias).

The model is trained through the train method of the class. It requires the following arguments:

train_rdd:	optional RDD to use as training set, if None, the method uses the entire data set as training set.
SGD:	boolean to indicate whether stochastic gradient descent should be performed.
SGD_pct:	Percentage of the training data that should be used in case SGD is true.
threshold:	threshold between 0 and 1 for the classification, if predicted probability > threshold, the email is classified as spam.

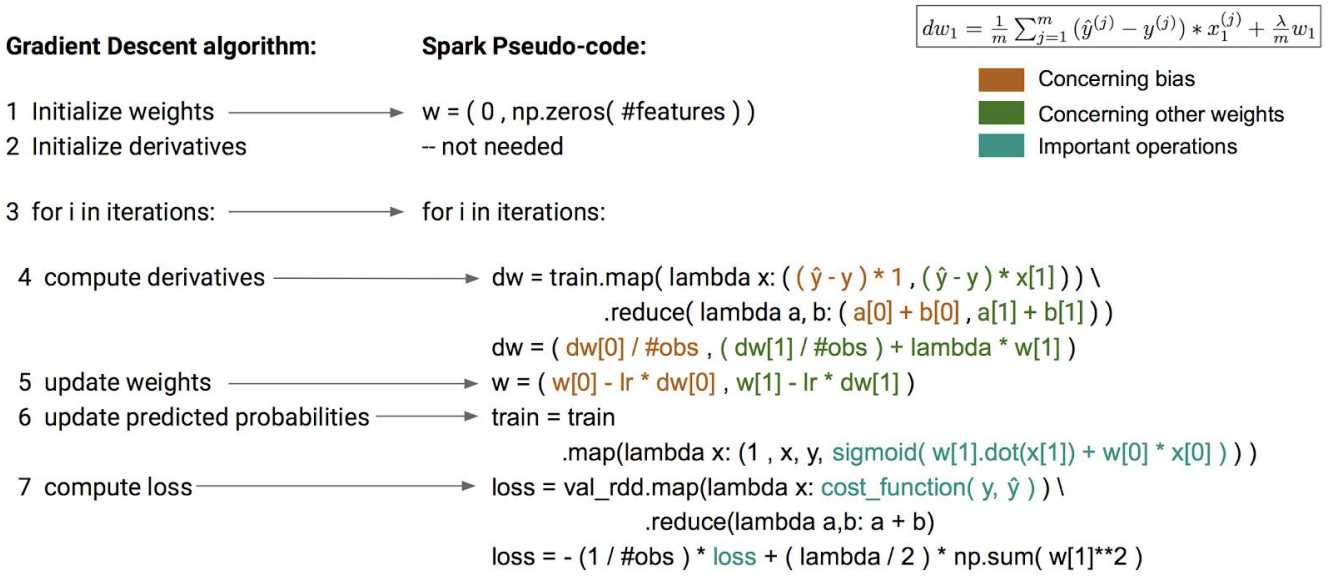
Having explained the degree of parallelism and the data structures used in the previous sections, the training algorithm with Gradient Descent is outlined below with the corresponding operations in terms of Spark. Please note, that the Spark algorithm is written in pseudo-code in order to increase readability. In the beginning, we thought we could put all the weights including the bias term into a single array of weights, however, the bias term should not be regularized, therefore we had to keep it separate, this now gets apparent in

the below pseudo-code. The parts in brown colour indicate the calculations for the bias term, while the green parts are for the rest of the weights and features.

Again, you can find again the previously described map and reduce steps for the calculation of the derivatives and for the loss (4 and 7). The predictions can be done in a single map step (6). Note that the operations marked with turquoise colour since it involves both the bias and the rest of the weights and it is the major operation in these steps.

In the beginning we experienced increasing time per iteration for the derivative and loss computation steps. We figured out that this was caused by the lazy evaluation of Spark and hence it did always recompute our entire training data set. Once we cached the training data set at the beginning of the iteration process, the execution times of the iterations were much faster and near constant.

The *train* method returns a dictionary with the necessary information of the model, that is, the performance measures and the learned weight data structure.



Validation

The validation procedure is implemented with a *validate* method in our *ParallelLogReg* class. The method takes as input:

val_rdd:	Test data set on which a model is to be validated.
w:	Weight vector of model to be validate.
threshold:	Classification threshold.
add_intercept:	Boolean to indicate whether the intercept column still needs to be added to the RDD.

The validate function calculates the predictions for the provided RDD based on the weight data structure and appends it to the tuples in the RDD. Probability prediction is done in the

same fashion as above in the training method. Subsequently based on the predictions, the loss can be computed with a map and reduce step as described above.

Lastly, the error metrics are computed with a map and a reduce step:

```
tp, tn, fp, fn = data.map(lambda x: f(x[2], x[4])) \
                          .reduce(lambda a, b: tuple(map(sum, zip(a, b))))
```

With the numbers of true positives, false negatives, false positives and false negatives, all the error metrics can be computed: Accuracy, F1, Precision and Recall.

Stochastic Gradient Descent

The stochastic gradient descent approach is a fairly simple extension to the normal gradient descent. In order to achieve a lower training time (not necessarily faster convergence) we can use a random sample of the training data at each iteration.

This requires resampling at every step and is implemented with the following condition:

```
if SGD:
    train = data.sample(False, SGD_pct).repartition(sc.defaultParallelism).cache()
    if i == 0:
        numObs = train.count()
```

We are repartitioning the resulting sample, so in case the partitions are of unequal size after the sampling. After this they will be balanced again and spread across all the workers, if there are as many partitions as workers.

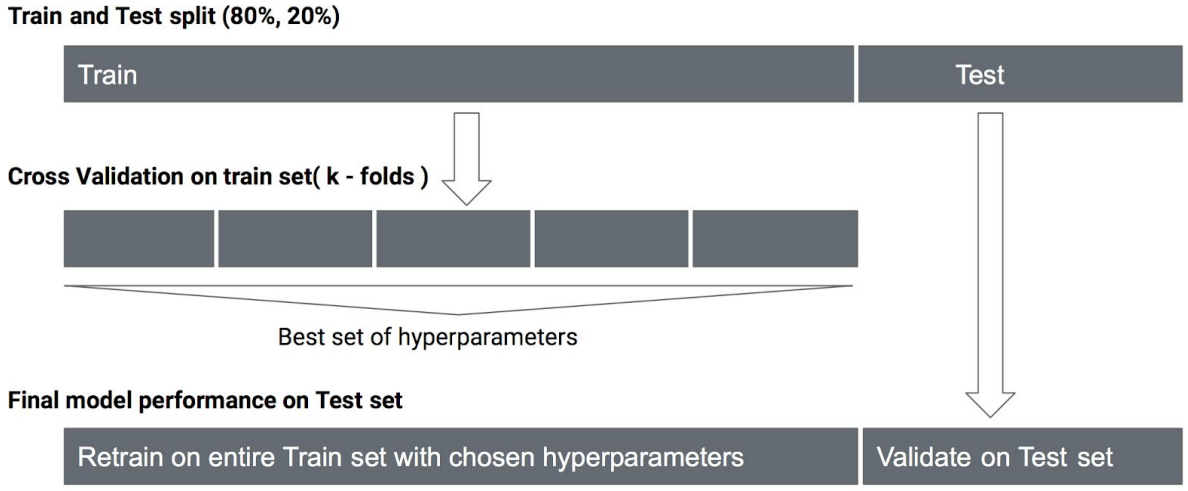
This resampling is done at the beginning of every new iteration of the GD process. As experiments showed, since we are training on a subset of the data, the time saving on training outweighs the cost of the resampling each time.

Cross Validation Procedure

The below figure illustrates the procedure for cross-validation. Before training anything we are splitting the data set in a training set (80%) and a test set (20%) which will be kept for the very end, in order to be able to estimate the generalization capabilities of the learned model. The test data set is never used for training or any kind of hyperparameter tuning. This is necessary in order not to bias the model.

Subsequently the training set is used for cross-validation in order to find the best set of hyperparameters, that is learning rate and lambda for regularization. Once we have found a good hyperparameter set, the model is retrained on the entire training set, because it doesn't make sense to train the model only on k-1 parts of the training data. One should always

utilize all of the data available for training in order to keep over-fitting to a minimum. The final step is to validate the learned model on the test set. The error rate and performance measures will estimate the performance of the model on completely unseen data.



Implementation

Since we have the *train* and *validate* method function defined in a way to be able to pass it a dataset to train and validate on, we extended our *ParallelLogReg* class by a *cross_validate* method.

There are two ways to parallelize the cross-validation (CV), first of all the training of a single model can be parallelized, as we have already done in the *train* method. Additionally, it would be possible to train entire models, that is the folds, in parallel. Say we have four workers available and we are training two folds, one could let two workers train one model and the other two workers the other model. However, we failed to implement this in Spark, since you somehow need to be able to package the model training as a task that you can pass to the respective workers. Spark is optimized to create these tasks itself, and we couldn't find a way to accomplish or circumvent this with a single SparkContext. Hence, eventually we went with the approach that the training of single models is parallelized as described above, but the k folds of CV are trained sequentially.

Experiments

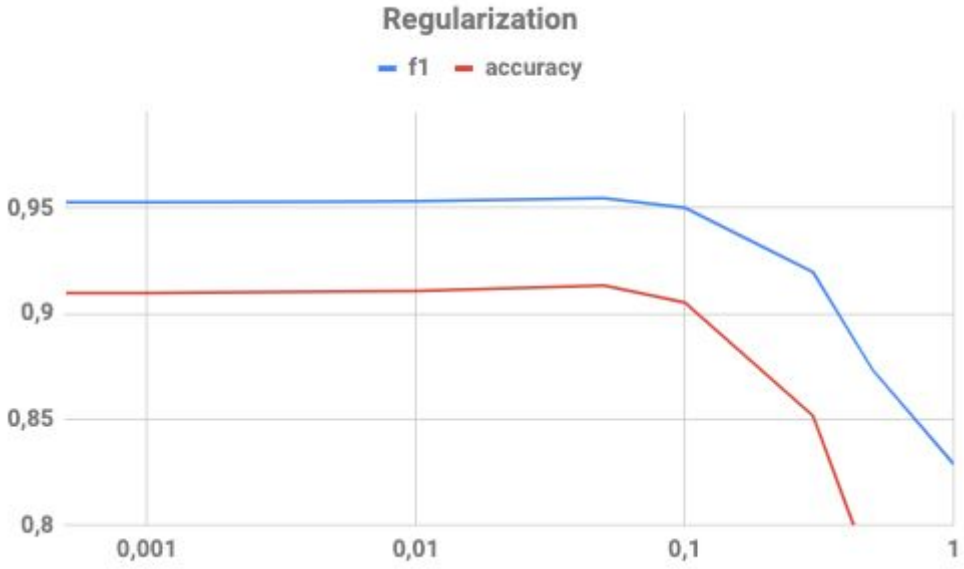
Setup

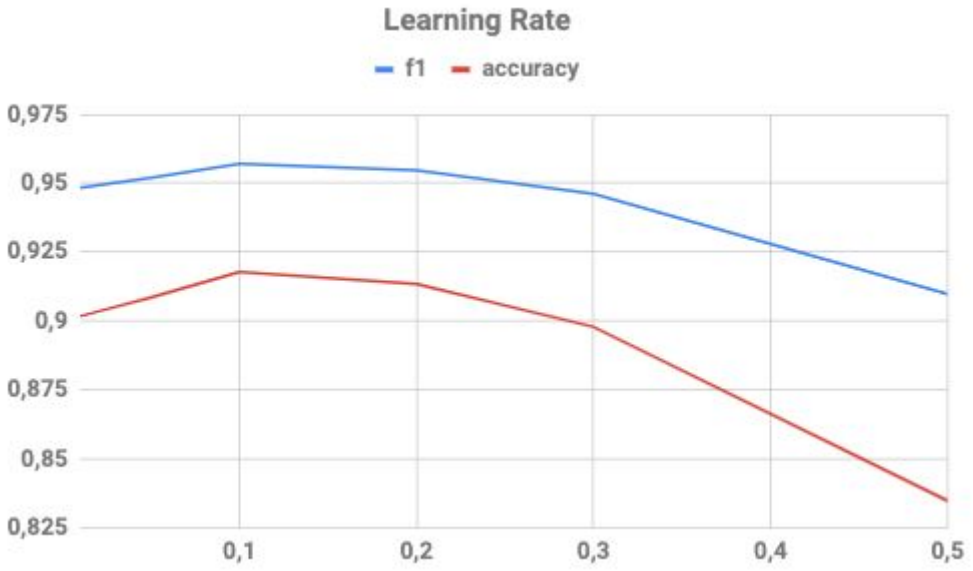
Experiments were run on a MacBook Air (13-inch, Early 2015) with a 1,6 GHz Intel Core i5 CPU with two physical and four logical cores and 8 GB 1600 MHz DDR3 RAM. Spark was run inside a Docker container with 2 GB RAM allocated and it takes as much CPU resources as it can get.

Hyperparameters tuning

To find optimal parameters we performed tuning with 10-folds cross validation with 50 learning iteration per each fold. We started with fixing the learning rate at 0.1 and run cross validation for different regularization values. The result of the experiments are shown by below charts. You can observe that for high values of regularization terms performance metrics are poor and model is underfitting. The highest accuracy and F1 score were achieved for 0.05 lambda regularization. Worth noting that performance does not worsen for really low regularization. It may be a result of the fact that, we trained our model on only 56 first degree features, consequently, model is simple and does not tend to overfit.

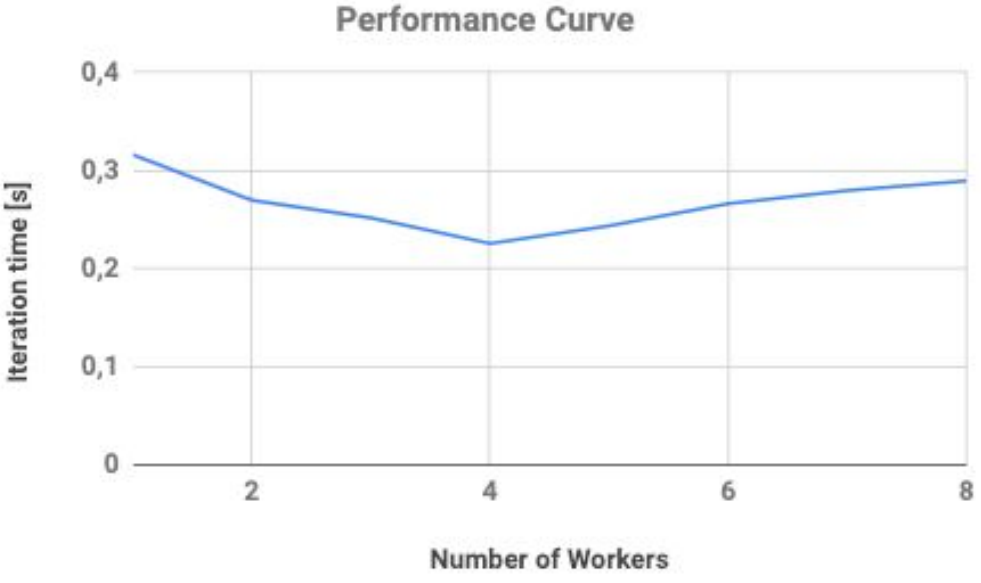
While the optimal value of regularization parameter was found, we continued hyperparameters tuning to find best performing learning rate. As shown on below graph, for learning rate higher than 0.3 model cannot converge and it results in poor performance. On the other hand, 200 iterations are not sufficient for learning rate smaller than 0.1. Eventually, our optimal learning rate is 0.1.

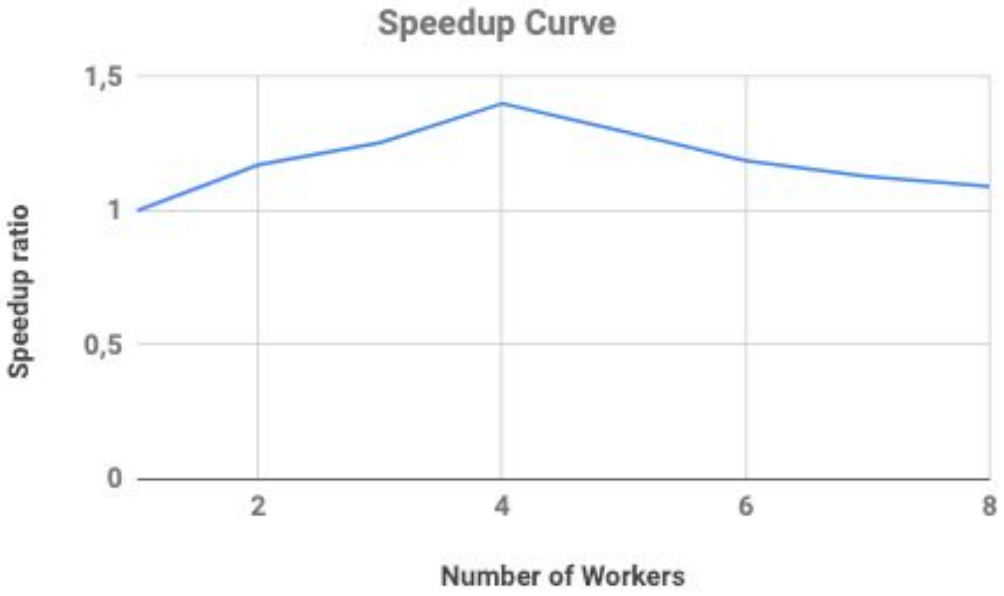




Computational Performance and Speedup

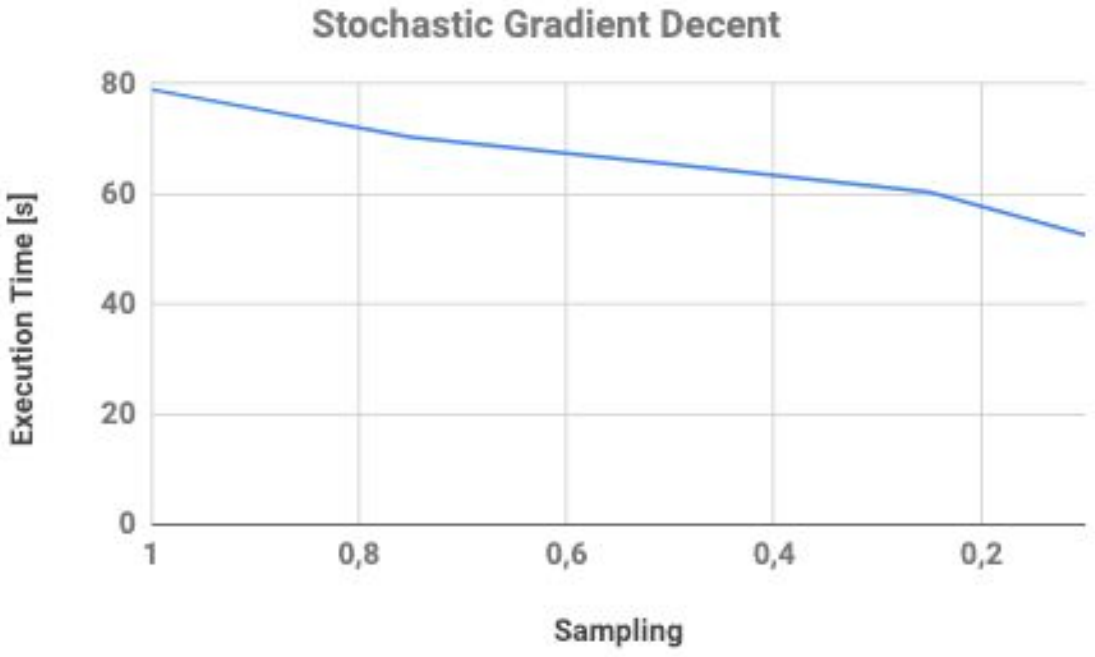
We started by measuring the computational performance in terms of speed of our implementation. The following experiments were run with 10-fold cross validation with 50 iterations at each fold, therefore, a total of 500 iterations. Average execution time per iteration for one worker is around 0.3s and as the graph below shows, adding workers increases performance up to four workers. With more than four workers, the overhead of communication and worker administration is outweighing the computational performance increase. However, with 4,600 observations we are using a very small data set, with a larger data set one would probably see more benefit of adding workers.

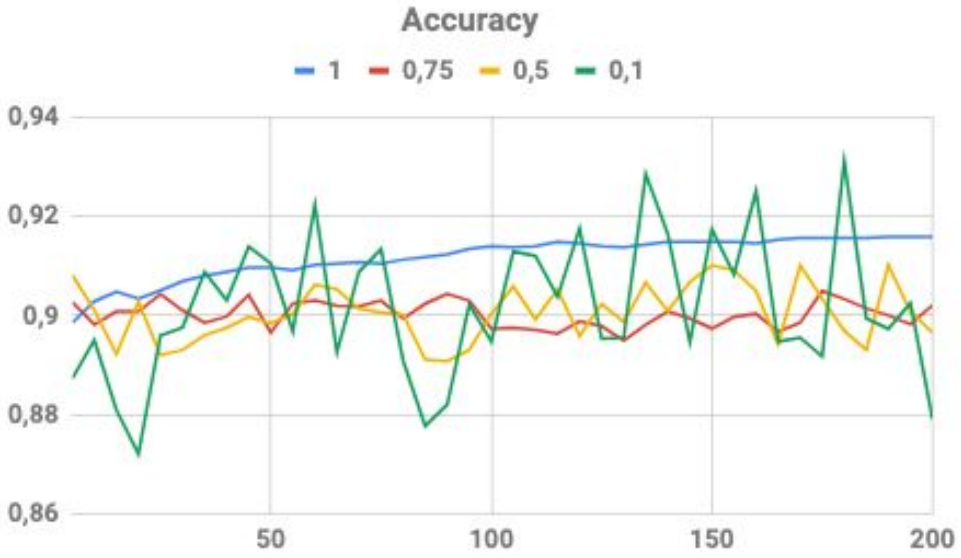




We also explored stochastic gradient descent as a method of speeding up learning process. It is called stochastic because samples are selected randomly from training set instead of taking whole training set in single iteration. Below chart presents how sampling decreased overall training time. It is important to note that throughout whole experiment, every 5th iteration model was evaluated against whole training set. Normally, this step is unnecessary and improvement in execution time would be even more significant.

Training accuracy of a model using stochastic gradient descent remains in 0.88- 0.92 for any sampling. The main difference is the smaller sampling the higher variance of training accuracy in consecutive iterations.



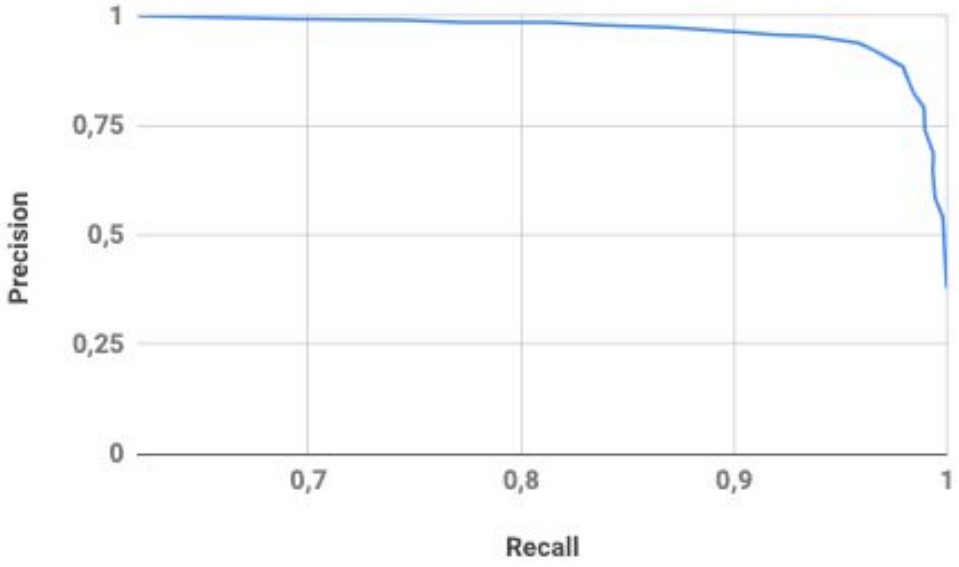


Model Convergence and Performance

Let's now look at the performance in relation to the number Gradient Descent iterations performed in order to get an idea about how fast the models are converging in terms of loss. The experiments were run with the best model from the hyperparameter tuning phase (lambda = 0.05, learning rate = 0.1). As one can see, the training accuracy is slightly higher than the validation accuracy, however the generalization is still satisfactory. The loss converged at about 100 iterations and there is no point in computing more than 150 iterations. Also 50 iterations already yield a low loss. Depending on the amount of data and available compute resources one could trade-off the running time versus model performance in a real world scenario.

The statistics of the optimal are:

In the Spam classification task we want to avoid classifying “good” emails as spam, that is, we want a high precision and we rather accept some spam passing through the filter. We can use the classification threshold to trade-off between precision and recall in order to get the desired compromise between the two of them. The graph below illustrates the tradeoff, where each point on the graph corresponds to a different threshold. Larger thresholds yield a higher precision and lower recall and small thresholds vice versa. What we can see is that even with really high precision we can still achieve an acceptable recall of around 70%.



Conclusion

The final performance of the model evaluated on test data - 20% of initial dataset, completely not seen during learning process is as follows:

Loss: **0.391**
Precision: **0.954**
Recall: **0.919**
F1: **0.936**
Accuracy: **0.880**

Model performance with low regularization indicates that one can increase the model complexity. Meaning, adding second, third order features.

We experienced significant improvement in execution time when we forced Spark Resilient Distributed Dataset to be cached.

Authors of the dataset state ~7% misclassification error which is comparable to our misclassification error which is 12%.