Parallel Implementation and Evaluation of Logistic Regression

Massively Parallel Machine Learning (2018-19)

Marcin Paszkiewicz
Moritz Meister

Data Set Description

Data set: **SPAM E-mail Database**

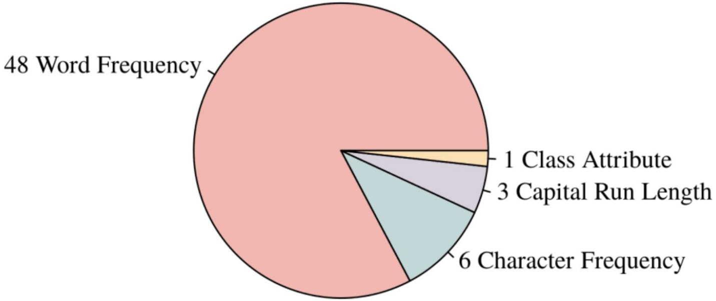
Origin: **Hewlett-Packard Labs**

Task: **Classification of emails as spam or non-spam**

Note: **False positives (marking good mail as spam) is very undesirable**
zero false positives in the training/testing set -> 20-25% of the spam would pass through the filter (HP labs)

Preprocessing: **Removal of column 57**
Standardization

Features:



Target variable:

Number of spam instances:	1,813	39.4%
Number of non-spam instance:	2,788	60.6%
Total of observations (emails):	4,601	

Definition: GD for Logistic Regression

Loss function to be optimized:

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

Derivatives:

$$dw_1 = \underbrace{\frac{1}{m}\sum_{j=1}^{m}}_{\text{2. Step}}\underbrace{\left(\hat{y}^{(j)} - y^{(j)}\right) * x_1^{(j)}}_{\text{1. Step}} + \frac{\lambda}{\cancel{m}} w_1$$

1. **Step:** **map, done in parallel**
2. **Step:** **reduce, across RDD partitions over all observations**

Implementation: data structures

Weight tuple:

Tuple(b , numpy.array([w1, w2 , w3, …, w56]))

Bias weight

Numpy array with the weights for the features for efficient vector operations

Main RDD:

RDD (Tuple(1, numpy.array([x1, x2, x3, … , x56]), y, ŷ, y_class)

Intercept term

True class

predicted class

predicted class probability

Numpy array with feature values for the observation

Implementation

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

Concerning bias
Concerning other weights
Important operations

Gradient Descent algorithm:

1 Initialize weights → w = (0 , numpy.zeros(#features))
2 Initialize derivatives → -- not needed

3 for i in iterations: → for i in iterations:

Spark Pseudo-code:

4 compute derivatives →
```
dw = train.map( lambda x: ( ( ŷ - y ) * 1 , ( ŷ - y ) * x[1] ) ) \
                .reduce( lambda a, b: ( a[0] + b[0] , a[1] + b[1] ) )
dw = ( dw[0] / #obs , ( dw[1] / #obs ) + lambda * w[1] )
```

5 update weights →
```
w = ( w[0] - lr * dw[0] , w[1] - lr * dw[1] )
```

6 update predicted probabilities →
```
train = train
      .map(lambda x: (1 , x, y, sigmoid( w[1].dot(x[1]) + w[0] * x[0] ) ) )
```

7 compute loss →
```
loss = val_rdd.map(lambda x: cost_function( y, ŷ ) ) \
                .reduce(lambda a,b: a + b)
loss = - (1 / #obs ) * loss + ( lambda / 2 ) * np.sum( w[1]**2 )
```

Cross Validation

Train and Test split (80%, 20%)

Train

Test

Cross Validation on train set(k - folds)

Best set of hyperparameters

Final model performance on Test set

Retrain on entire Train set with chosen hyperparameters

Validate on Test set

Stochastic Gradient Descent

Goal: **Stochastic approximation of normal Gradient Descent.**

Advantage: **Faster training iterations, since training is done on a random sample of the entire training data.**

How? Define what percentage of the training data should be used for the actual training.
Resample before every new iteration step.

```
for i in iterations:
    train_sample = train_data.randomSample( withReplacement = False, percentage )
                                  .repartition(sc.defaultParallelism)

    continue as usual with train_sample
```

Experiments

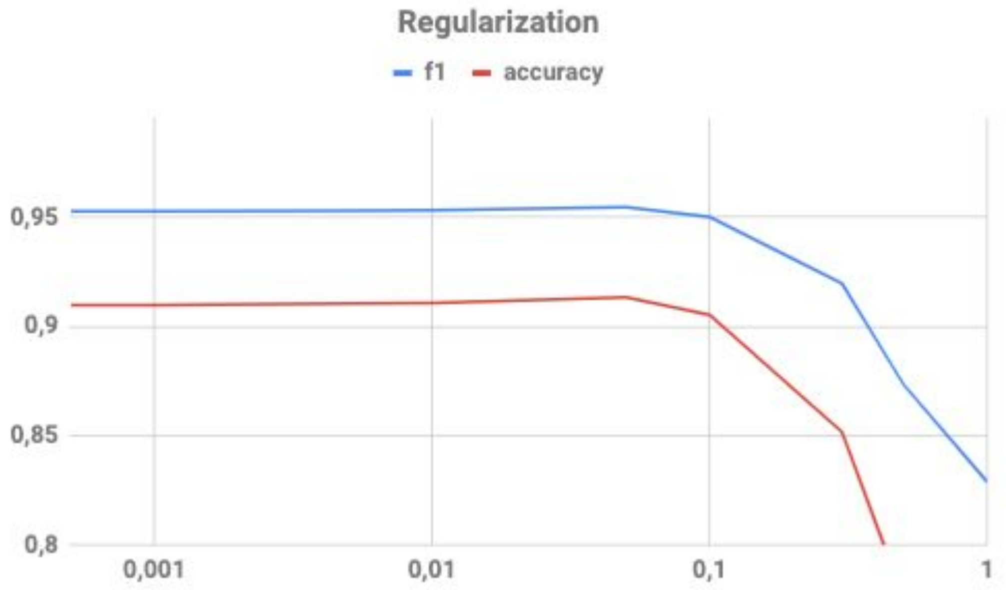
Hyperparameters tuning:

Regularization: 0.05

Cross-validation:

10-fold, 50 iterations per fold

Learning rate: 0.1



Experiments

Hyperparameters tuning:

Learning rate: 0.1

Cross-validation:

10-fold, 50 iterations per fold

Regularization: 0.05

Experiments

Training parameters:

Iterations: 200

Regularization: 0.05

Learning rate: 0.1

Final model test validation metrics:

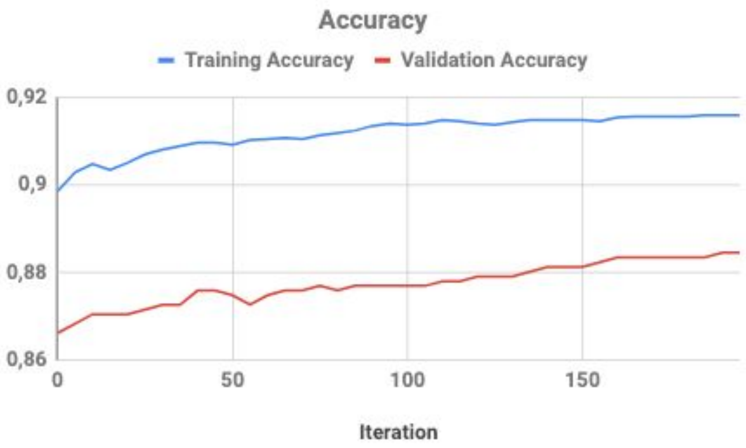
Loss: 0.391

Precision: 0.954

Recall: 0.919

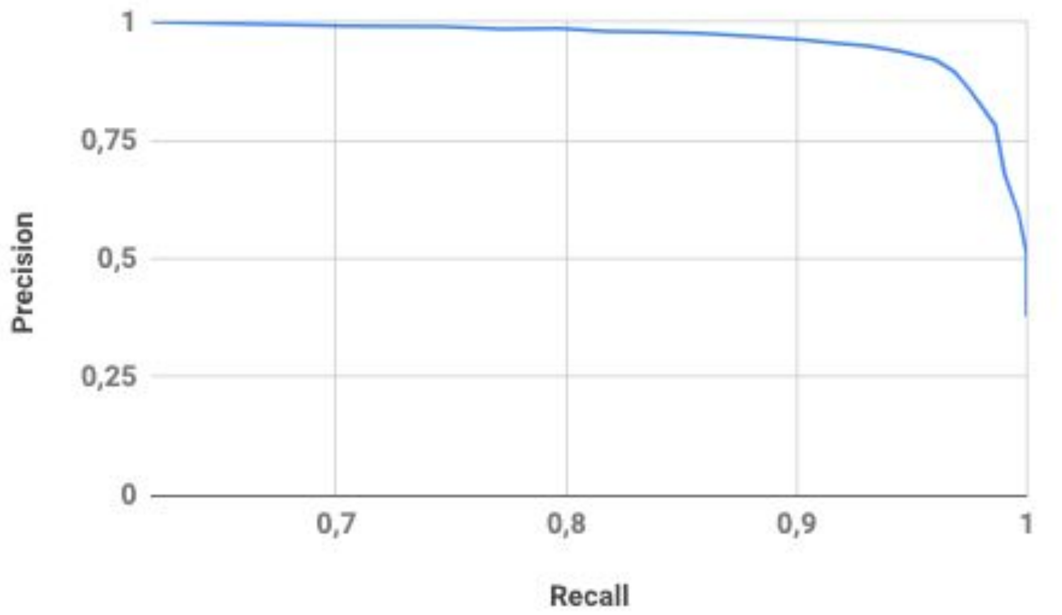
F1: 0.936

Accuracy: 0.880



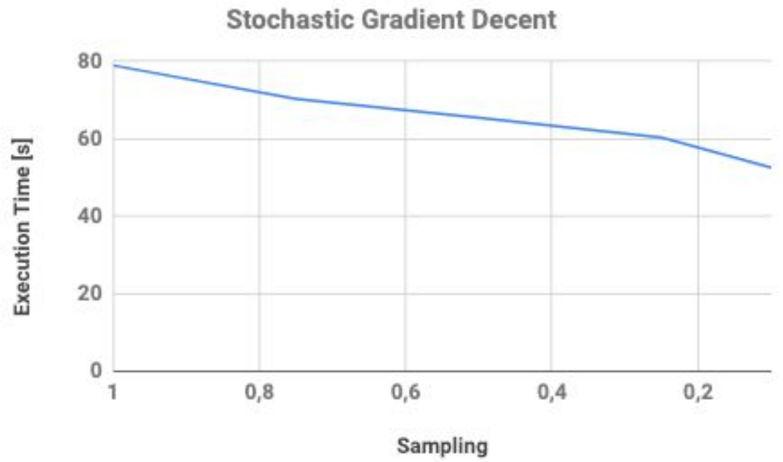
Experiments

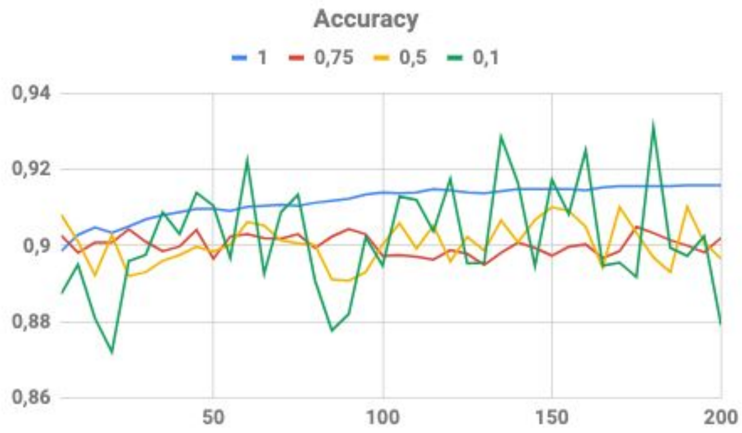
By adjusting prediction threshold we can steer towards precision or recall.



Experiments

Training time decreases as expected with smaller samples.

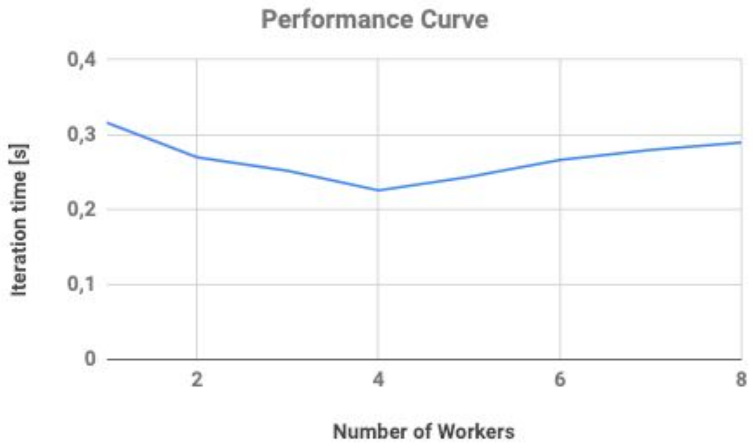


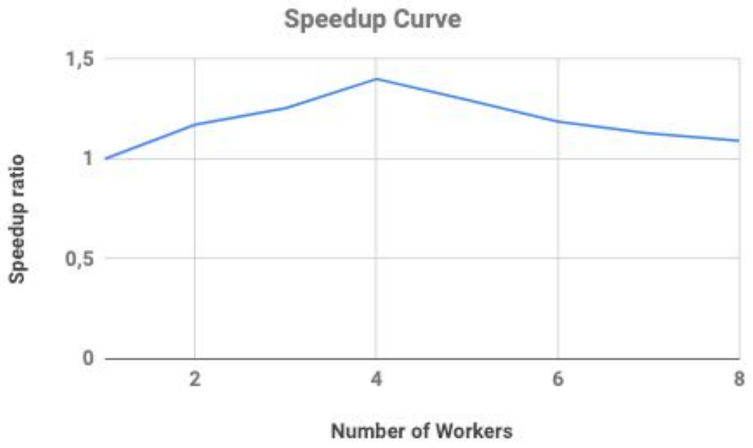


Experiments

Overhead for more than four workers outweighs the performance increase.

Based on 10-fold cross-validation with 50 iterations each, that is 500 iterations in total.





Conclusions

Speedup:

Caching RDDs makes a big difference.

Model complexity:

Good performance with low regularization, indicates that the model could be more complex.