ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Machine Learning with Logistic Regression Study

Korovesis Panagiotis	1115201700056
Korovesis Charalampos	1115201700057
Koumertas Stavros	1115201600231

Instructor: Paskalis Sarantis

28 January 2021

1. Introduction

While attending the course “Software Development for Information Systems” we were asked to develop a machine learning program with the purpose of determining whether two cameras, taken from a wide range on sites, are actually referring to the same physical product.
Some of the core demands were storing a large number of records while being able to quickly access any given one, creating a machine learning classifier and training it, in order to test the trained model on a set of unknown data.

2. Implementation

i. Pre-processing

The data was in the form of json files distributed in folders. In order to retrieve the required data from each file, a parser was created. We created a static Hash table to store the information, whose size was adjusted to the optimal value for the data volume.
The essential information of each file resides in the product description. Following basic machine learning principles, while parsing each file, we decapitalized all letters and formatted the text removing:

- Non-ascii characters.
- Punctuation characters.
- English stopwords.
- Words with length of 1.

The description is then stored separately for each file in the form of a unique words list and then added to the Hash Table.
A Hash Table was likewise used to store all the unique words in the data, along with the frequency of each one.
Then by parcing a given file we implemented positive and negative correlations for some of the files.
The unique words in all of the data set were over 100.000. Thus, we decided to remove words with a tf-idf score lower than a set threshold. From the definition, terms with poor tf-idf score appear frequently in the data set, thus providing insufficient information. In order to achieve this, we calculated a tf-idf value for each word originating from the average tf-idf values of the word as it was calculated for each file.
(word tf-idf = sum(word tf-idf from each file) / number of word appearances)
Having the final set of words, we represented each file as a tf-idf matrix. To avoid the large number of zeroes, we used sparce matrixes for each file.
Finally we created a logistic regression model, which we trained.

ii. Training

We use the logistic regression statistical model to train the classifier. 60% of the available data is used for the train process, 20% for validation and 20% for the final test set. The train data is split in batches before being “crunched” from the model.
A single train with one batch works as shown below:

```
For each weight:
    weight -= gradient-value()
```

The gradient value is calculated from the whole batch with the following method

```
for each matrix in the batch:
    get the value with the same index as the weight
    if value is not zero:
        errorSum = gradient-descend-derivative()
    return errorSum * model-learning-rate
```

We train the model with batches to avoid steep changes caused by the single step training method.

To achieve the whole training process, the following algorithm is used:

```
threshold = initial_value
training_set = initial_training_set
while (threshold<0.5):

    b = train_model(training_set)
    for all x: // all pairs
    if (p(x,b) < threshold) or (p(x,b) > 1 - threshold):

        add(x, p(x) new_training_set)

    training_set = resolve_transitivity_issues(training_set, new_training_set)
    threshold += step_value
```

The threshold is initially set at 0.10 and then adjusted to 0.15. The train_model() is executed three times. Once with the initial set and twice with the enlarged one. After each training, metrics from the validation set are displayed to track the training progress.
In order to resolve possible transitivity issues, we have created a new Hash Table containing only the initial train data. In addition, the pairs that meet the threshold, are placed in a tree according to the model prediction. We perform an inorder traversal -so the pairs are sorted- and if the current doesn’t contradict the existing correlations in the Hash Table, we insert it. If it’s not possible we dismiss it. After this process is completed, the valid pairs are also inserted in the train set. It’s important to note that the pairs mentioned above are derived from the entire

data, by traversing the initial Hash Table

iii. Synchronous Gradient Descent

The size of the data resulted in long train and run times. To resolve this issue, we implemented threads to the training process via a thread pool.
A job scheduler was implemented to synchronize threads with jobs. Jobs can be any functions with any arguments. The job scheduler initializes the threads requested by the program and assigns to each one a thread function. It then creates a queue where jobs will be placed while also holding the required mutexes and conditions for the communication with the threads. The threads are “sleeping” while there is no work to be done. When a job is added to the queue the threads are notified. When a thread receives a job, it extracts and executes the function inside, then checks the queue for a new one. If there is none it goes again back to sleep. The job scheduler waits for the treads to terminate, then the program proceeds.
In the training process, we use the thread pool to calculate the new weights for the classifier. The initial batch size is split depending on the number of available threads. Jobs are created with the correct arguments for each thread such as start and finish parameters. The training process is synchronous because the last thread, to calculate the error required, is also the one to return the new weight to the classifier. In other words, the program must wait for all the threads to finish before updating a weight.

iii. Testing the classifier

After the training is completed, we test the classifier with the test set and we print the appropriate metrics for the evaluation of the progress. Those are the precision, recalls and f1. Finally, all the used memory is freed and the program exits.

3. Experiments

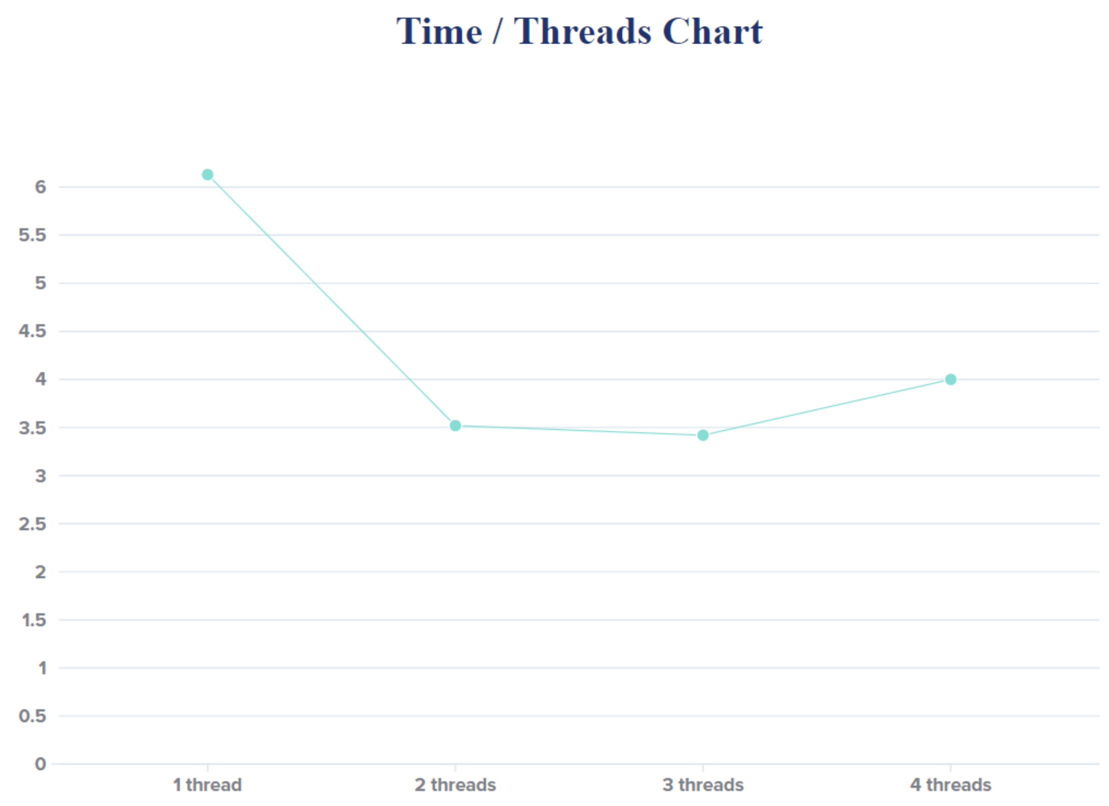

Figure 1 Execution Time depending on the number of Threads used

The example shown in the chart uses a sparse matrix of ~ 1000 spaces for each json file. We can clearly see that using threads has drastically reduced the time required for the completion of the program. Time is increased when using 4 threads due to the limited capabilities of the computer used to run the program. The optimal number of threads for this machine is three.

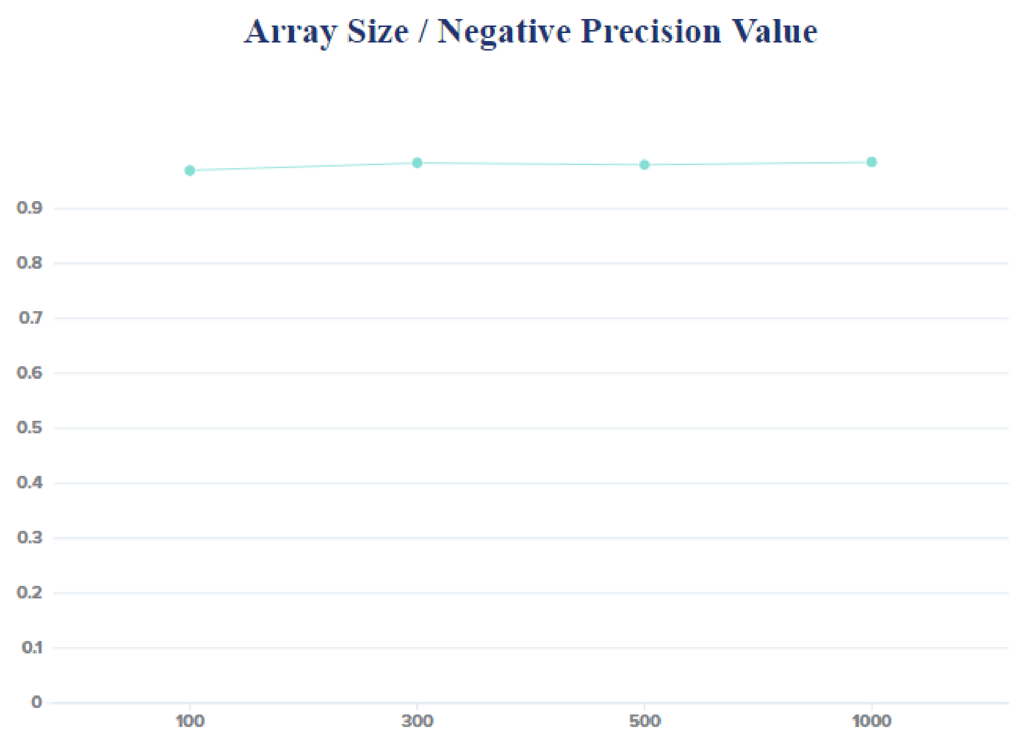

Figure 2 Precision for Negative Pairs with different matrixes

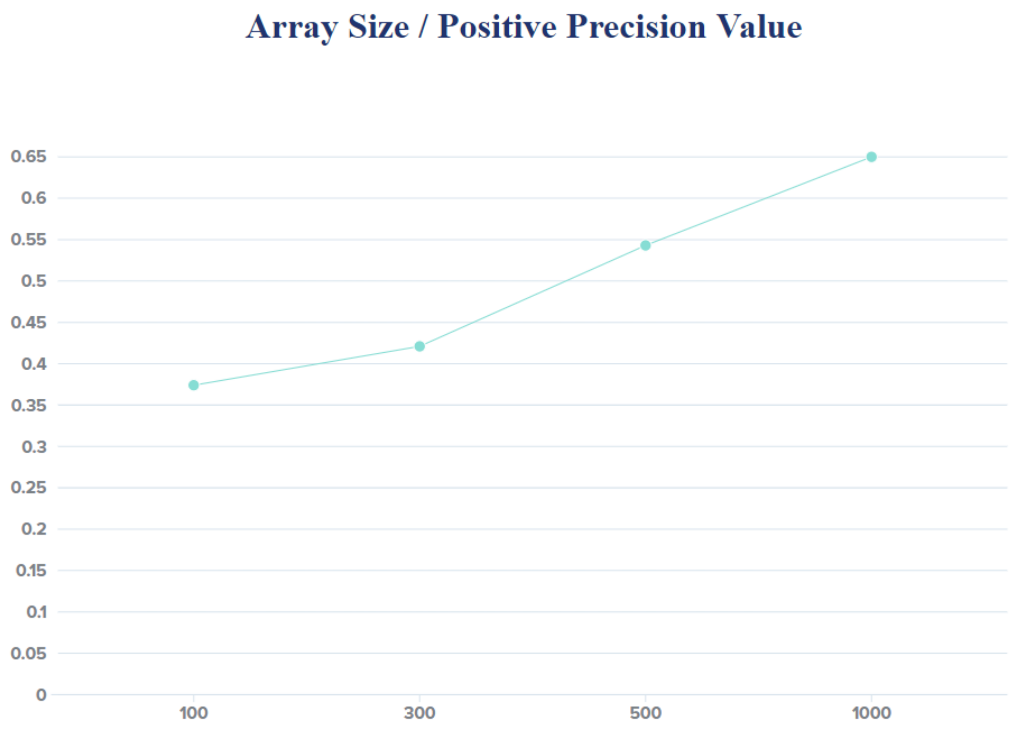

Figure 3 Precision for Positive pairs with different matrixes

The diagrams above, test the importance of the matrix size -used to represent the json files- on the accuracy of the model. We can observe that the size doesn't play a significant role for the negative pairs, however there is a steady improvement for the positive pairs as the size of the matrix increases. This phenomenon is caused by the train set, which is heavily biased towards zero. As a result, the model learns to recognize negative pairs much better than positive ones.

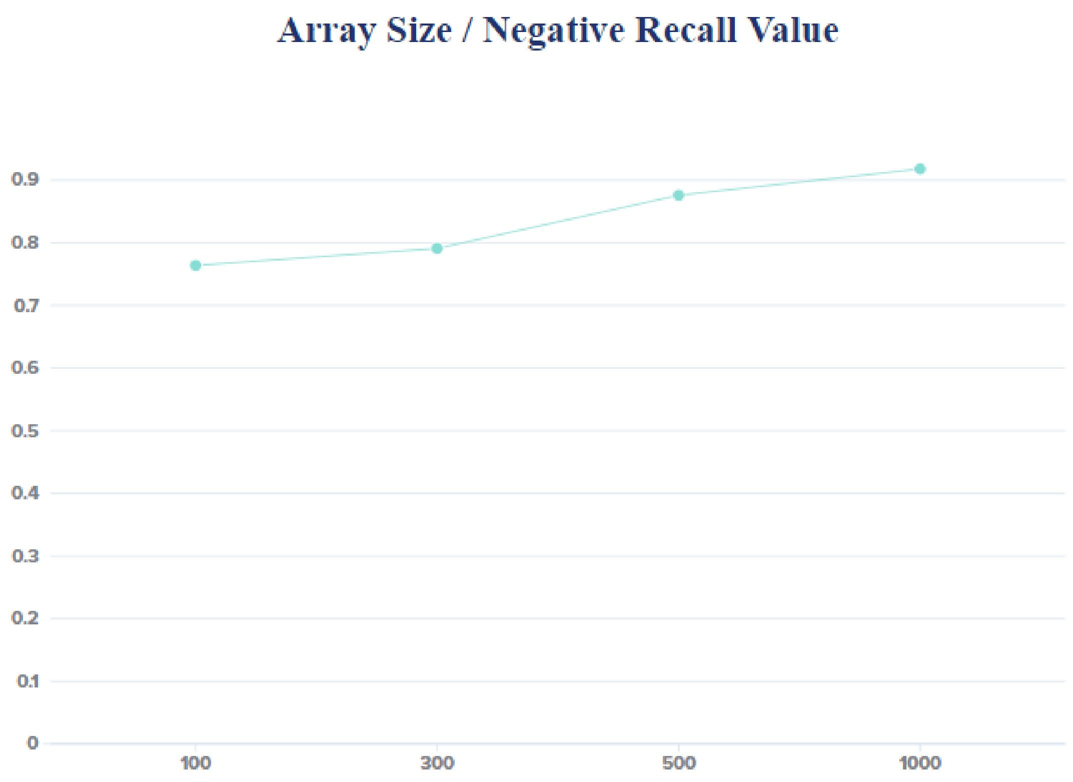

Figure 4 Recall value for Negative pairs with different matrixes

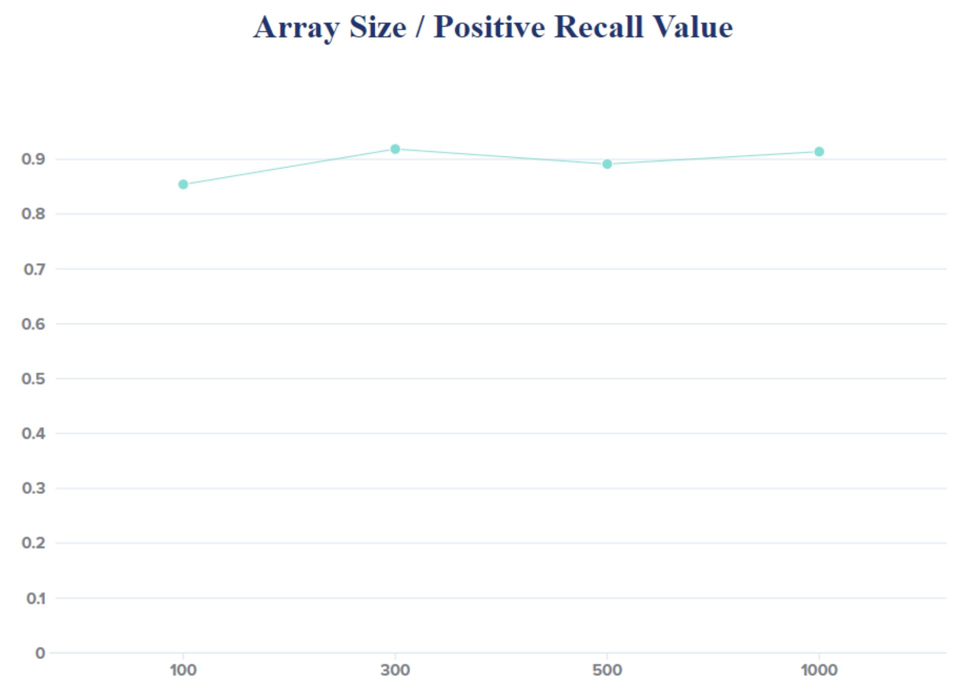

Figure 5 Recall value for Positive pairs with different names

The diagrams above show the fluctuation of the recall value. Recall is the fraction of relevant instances that were retrieved. We can observe a relative stability regardless of the matrix size.

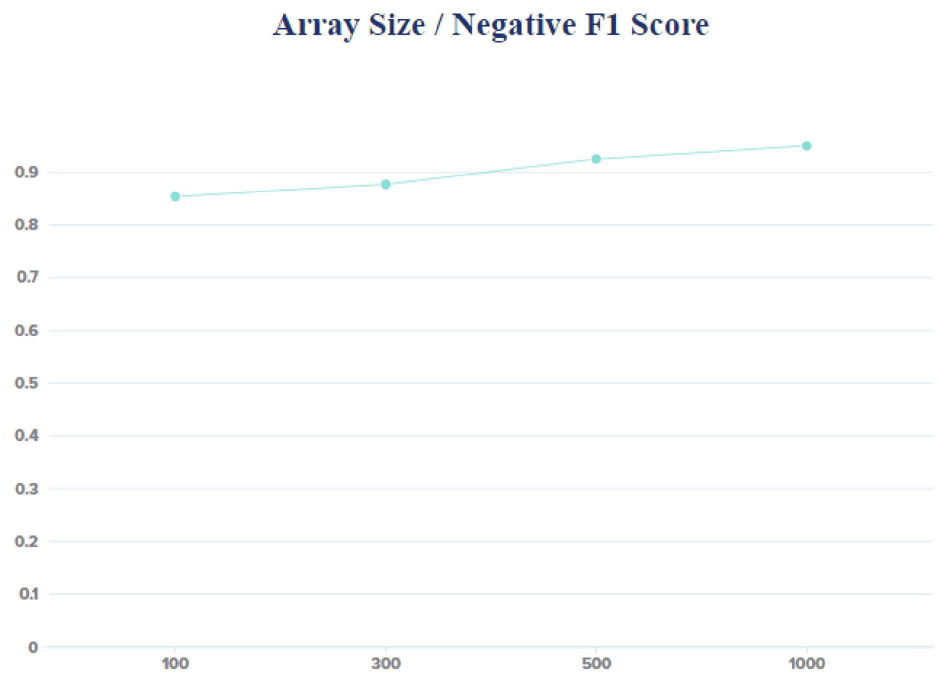

Figure 6 F1 score for Negative pairs with different matrixes

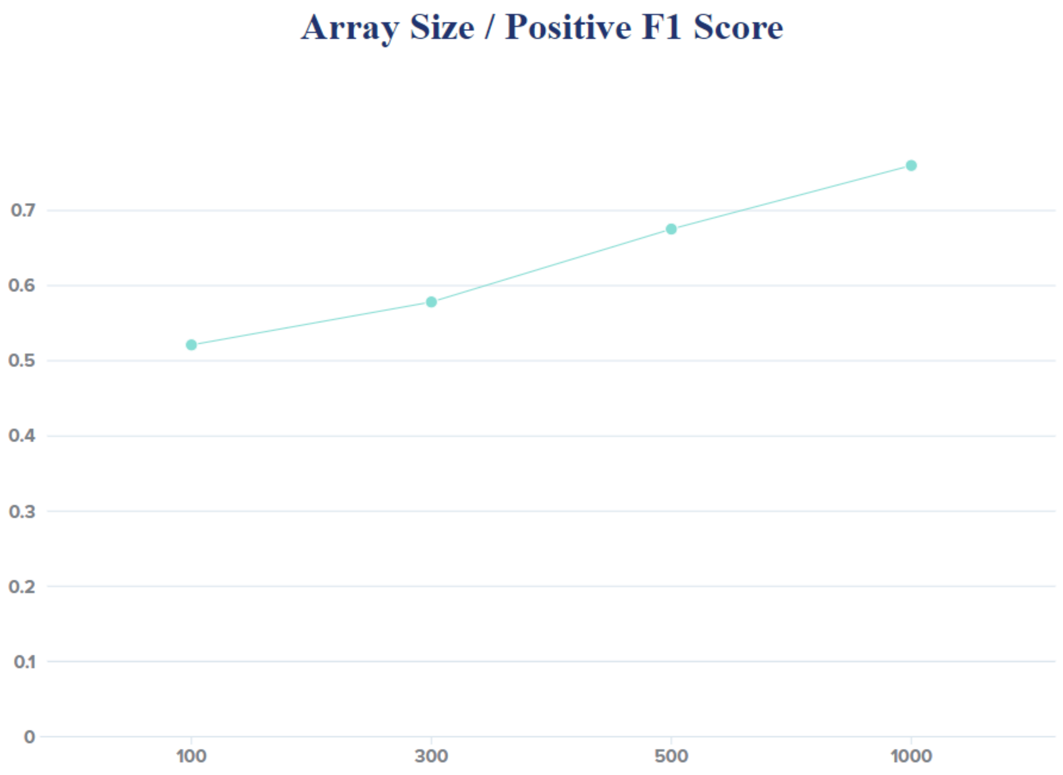

Figure 7 F1 score for Positive pairs with different matrixes

The diagrams above show the F1 scores for different json matrixes. The F-score is a measure of a test's accuracy. It is calculated from the precision and recall of the test. The same pattern can be observed here as in the precision diagrams.

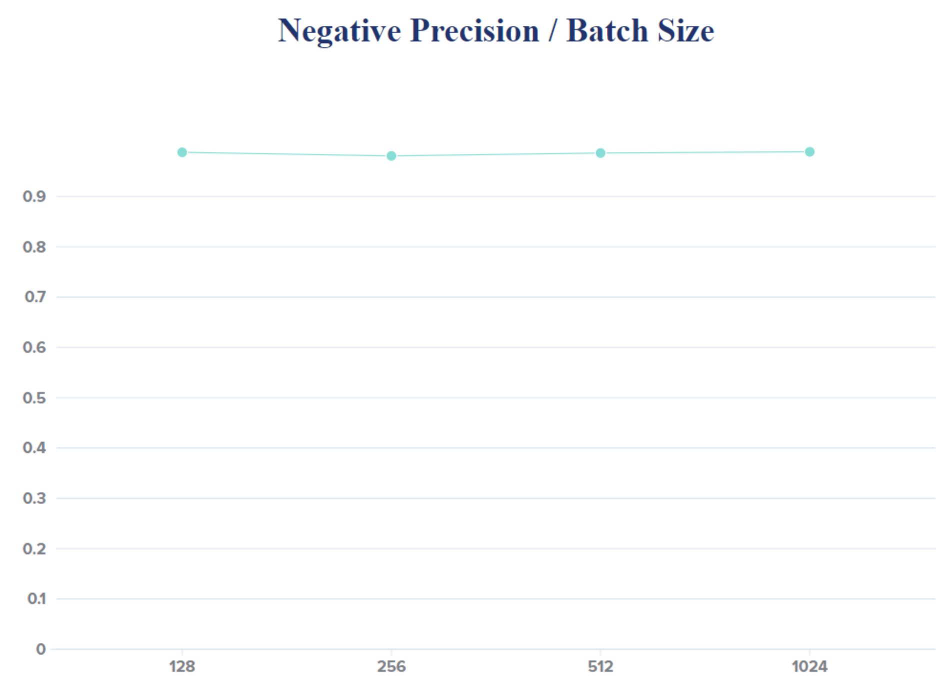

Figure 8 Precision for Negative pairs with different batch sizes

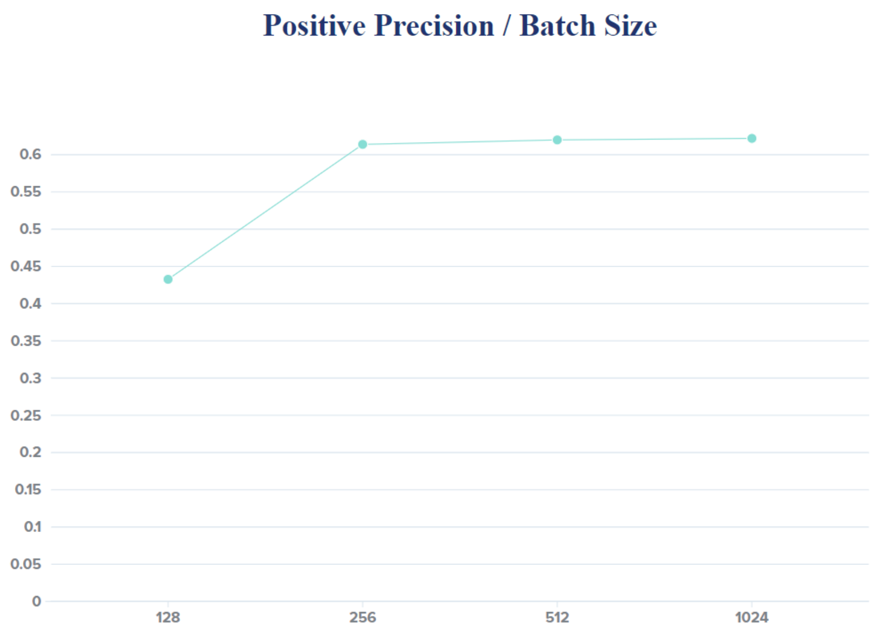

Figure 9 Precision for Positive pairs with different batch sizes

The diagrams above show the set accuracy when using different batch sizes in the train process. We can observe that scores for negative pairs are not affected from the batch size used, whereas positive pairs have a significant precision increase for batch sizes over 128.

4. Conclusion

Having completed this project, we can now present the following observations.
In an ideal balanced set, the size of the matrix used to represent the data can greatly affect the classifier results. As can be seen in the graphs above, larger matrixes lead to better training and subsequently better predictions. The same can also be said about the batch size used for the training itself. This behavior is due to the fact that when using larger batches, radical shifts of a certain weight value can be avoided as the new weight value is calculated for all the elements in the batch. Last but not least, thread use is of vital importance, as the training process is largely based on read-only operations, making threads ideal for parallel execution, resulting in much smaller run-times.

References

Logistic Regression Model
https://en.wikipedia.org/wiki/Logistic_regression

Graph Designer Tool
https://spark.adobe.com/

Assignment
https://bit.ly/2Mt8HEZ

Jianmin Chen, Rajat Monga, Samy Bengio & Rafal Jozefowicz
static.googleusercontent.com/media/research.google.com/en//pubs/archive/45187.pdf