BDPP Project Report: Handwritten Digits, Classification task

Johannes Hägglund

May 2021

1 Introduction

Image classification is the task of extracting information from labeled images, build and train suitable Machine learning (ML) models which then can be used for different applications. Image classification is widely used today within applications such as object recognition, image retrieval, medical diagnosis, fingerprint recognition etc [1].

As described and introduced, this project is about image classification, and to specify, using the MNIST handwritten digits [2] data-set to train, test and evaluate different types of suitable ML models. The data-set comes in two different folders, one for training data, and another folder for test data. Each folder contains image folders for each label, and image labels are ranged from digit 0 to 9. The digits are 28x28 gray scaled images and the training set contains 60 000 different images while the test data contains 10 000 different images. Figure 1 shows on 100 randomized digits from the dataset.

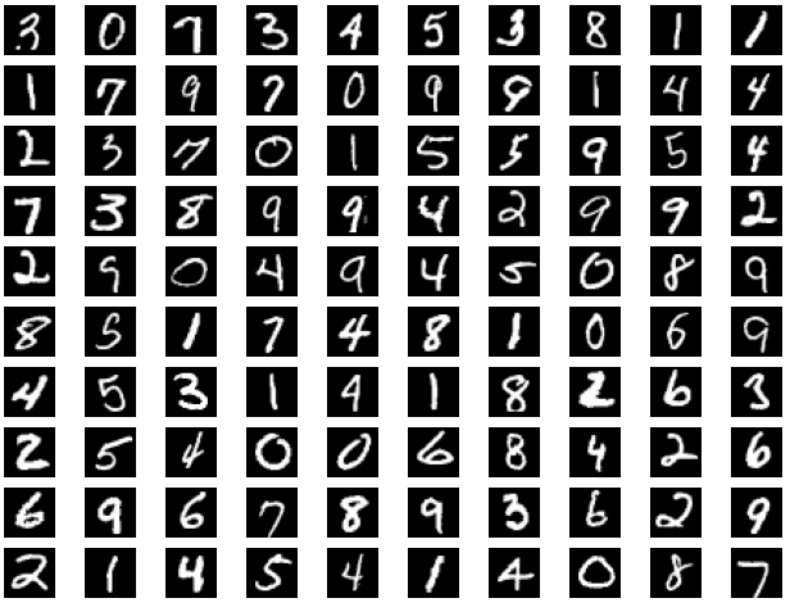


Figure 1: Plot of 100 random digits

The problem will be solved by applying suitable machine learning models within the category for multi-class classification, which is where the input can be classified between two or more classes, and in this case it will be digit 0 to 9. There are some ML models that is suitable and that can be used to solve this type of task, the chosen models in this project are Naive Bayes, Logistic Regression, Random Forest and Multi-layer Perceptron Neural Network.

The purpose of this project is to examine the advantage of using Apache Spark that uses parallelism over more distributed computers/cores to achieve better performance and lower computation time, especially when working with big data.

The goal is to implement the described ML models using Apache Spark and compare MLlib pipeline approach over MLlib RDD approach.

2 Method

2.1 Data Preprocessing

Before implementing and train ML models it is important to pre-process the data. This because the data can come with a lot of missing values, noisy data, values that range from small to big numbers and unbalanced data, all this can affect the ML model and is therefore crucial to examine and fix before building the models.

The data used in this project is mostly pre-processed, the images are already of gray-images, scaled (values ranges from 0-1) centered at the origin (mean=0), and cleaned. But some important techniques to mention are image denoising, image scaling and convert to gray-scale. The purpose of image denoising is to remove the noise from the image that could possibly remove the content. Image scaling, can be done by either normalizing the image which will scale the pixel values to the range 0-1, or pixel centering which will scale the image such that the pixel values have zero mean, and image becomes centered around the origin. Colored images will increase the computational requirements, and also the complexity of the model, to avoid this we use gray-scaled images when we train and evaluate the ML models.

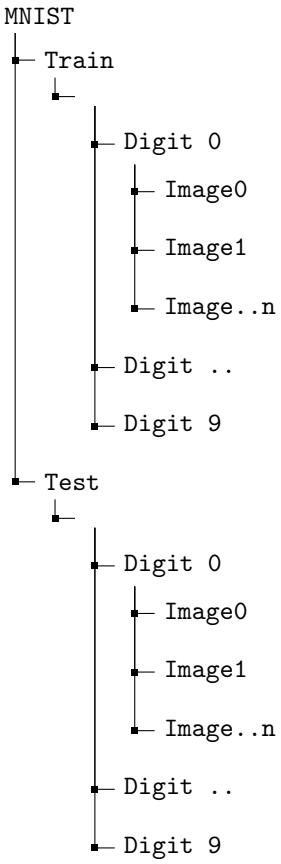


Figure 2: The tree of the directory

Figure 2 shows the structure of the directory MNIST. The preprocessing in this step will be to iterate through each folder, and for each image, convert it

- Master Node (1 master, N workers):
  Machine type: n1-standard-8
  Primary disk size: 30 GB
- Worker nodes:
  Machine type: n1-standard-4
  Primary disk size: 30 GB
- Optional Components: Jupyter Notebook

Figure 3: Google Cloud config

to one 1-D array containing 784 features (28x28 pixels). The final array will be of dimension 2D where each row will contain 785 values, 784 for the pixel and the last value will be the label for corresponding image. Finally the data will be written to two csv files, train.csv and test.csv.

To be able to train and evaluate the models it is required to transform the features into a single vector column. This can be accomplished by the transformer *VectorAssembler* which can be imported from the library **pyspark.ml.feature**.

Due to the big amount of data the training set have, this could possibly affect the training time a bit. Therefore two solutions will be provided, one where the training, testing and evaluation is done with all the data, and another solution where it is using a minimized version of the data. The minimized version will be implemented such that it samples a specific amount of data-points from each label, and in this case it wont result in unbalance data. Just by dividing the data by the half and choosing the first half could result will probably result in unbalance data and in that case the models will be over-fitted to one label, the majority.

2.2 Google Cloud

To measure performance and analyze the results, the code will be running on the cloud where the hardware can be configured by the users own requirements, though the better hardware, the more the user will have to pay for the usage. Before we can start to run the code we have to create a project, create a bucket where we can store the train and test data, and also an cluster. For this project, the configurations for the cluster can be shown in Figure 3.

2.3 Spark Implementations

n this section we talk about two The ML models will be implemented using two different approaches, MLlib Pipeline which works with DataFrames and MLlib RDD which works with Resilient Distributed Datasets. Both of these approaches will be compared in computation time and achived accuracy. To find the best

parameters for each model, that gives the highest accuracy, hyper-parameter tuning will be applied on each ML model. After some research I found that libraries for hyper-parameter tuning is not available for the MLlib RDD. Since the main goal of this project is to compare computation time and accuracy between the models, there will be no hyper-parameter tuning for the RDD approach. Instead the best parameters will be found by tuning the pipelines, and then these parameters will be used to initialize the RDD models.

2.3.1 MLib Pipeline [3]

Pipelines are commonly used to easier combine steps into a whole workflow instead of doing all steps one by one, line by line. The advantage of pipeline is that it only requires to be initialized with given stages. These stages are often preprocessing steps combined with the ML model. Since this project does not involve any preprocessing steps, the pipeline will only include one stage, the ML model.

The ML models will be imported from the pyspark library **pyspark.ml.classification**, each model will be initialized with random parameters which could cause a bad first accuracy, but later on the tuning will try to find the best values for each parameter. The tuning step will be accomplished using pysparks *Crossvalidator* which is imported from **pyspark.ml.tuning**. Also an additional class-object that will be imported is the *ParamGridBuilder* which lets us to specify what values each parameter can take, from these values the best one will be chosen. The evaluation of each model will be done using *MulticlassClassificationEvaluator* from **pyspark.mllib.evaluation**. In this project the accuracy will be calculated of each model, the evaluator can be initialized to evaluate the accuracy by specifying metricName='accuracy'.

2.3.2 MLlib RDD [4]

The RDD based ML models will be implemented using the MLib RDD-based library from pyspark. The difference here from the pipeline approach is that the Multilayer Perceptron Neural Network model does not exists for RDD and can therefore not be compared, and the data can not be in the same format. RDD based models requires LabeledPoint for each data-point, and can be imported from **pyspark.mllib.regression**. LabeledPoint takes two arguments as input, the first is the label and the second is the feature-vector. All rows in both train and test data will be converted into this format.

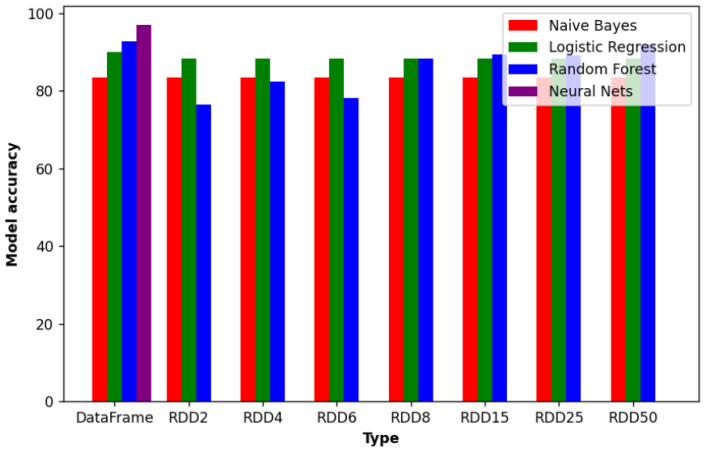
As mentioned before, the best parameters from the tuning library will be used here since there is no tuning for ML models based on RDDs, each model will be timed during training, testing and evaluation, this with different partitions. The partition of a RDD can be changed by either using *repartition(numPartitions)* or *coalesce(numPartitions)*.The difference between them is that *repartition* is able to increase and decrease partitions, though this function requires to shuffle the data, which can be computational expensive. Coalesce is only able to decrease the number of partitions which in some case can be a problem, e.g.

if current partitions are 2 and we want to change it to 5, this can not be accomplished with *Coalesce*. Though *Coalesce* does not need to shuffle the data which can be an advantage. Finally the RDD solution will be compared to the pipeline by execution time and accuracy.

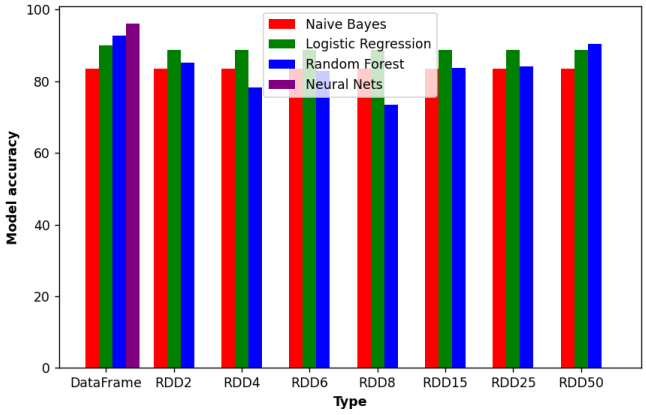
3 Results

3.1 Accuracy and time comparison

The results is obtained by comparing DataFrame and RDDs in both accuracy measure and time measure. Due to the big amount of data that the dataset contains, comparison is done by running all models both with all data and a minimized version of the data.



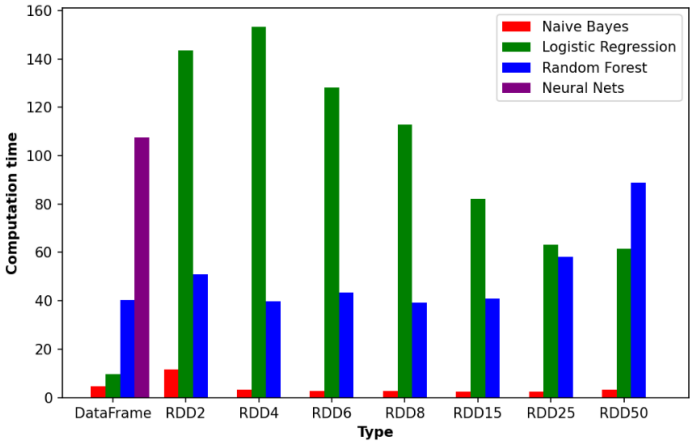
(a) Obtained accuracy with all data



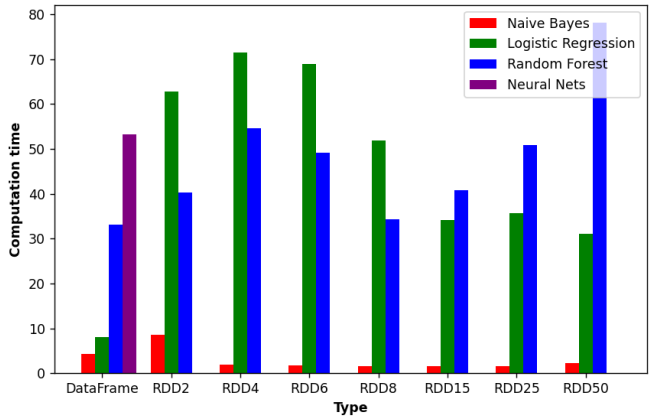
(b) Obtained accuracy with minimized data

Figure 4: Comparing accuracy, both for all data and minimized data

Figure 4 shows the results obtained in the accuracy measure, where subfigure 4a shows on the accuracy score with all the data used and subfigure 4b displays the accuracy score with the minimized version.



(a) Time comparison when using all data



(b) Time comparison when using minimized data

Figure 5: Comparing time, both for all data and minimized data

Figure 5 summarizes the obtained results and comparisons, both when using all the data and the minimized version. Subfigure 5a contains the time taken to build the models using all the data while subfigure 5b contains the obtained result using the minimized version.

3.2 Hyperparameter tuning

3.2.1 All data

Hyperparameter tuning was performed on each model in purpose to achieve higher accuracy score. The Naive Bayes model was tuned using 11 different

regularization parameters between 0 and 3. It took 54 seconds to improve the model from 83.57% to 83.59%.

Logistic regression was tuned with the regularization parameters [0, 0.05, 0.08, 0.1, 0.3], max iteration 10, it took 67 seconds and it still obtained the same accuracy as the first model with 90%.

Random forest was tuned with 2 different values of trees, 5 and 100. Also the maxdepth was tuned, and the model was able to improve from 77% to 92.75%, with a tuning time on 187 seconds.

Multilayer perceptron neural network took 396 seconds to improve, and improved in accuracy from 84% to 97.06%, this by tuning the attributes, layers and max iterations.

3.2.2 Minimized version

The minimized version did almost got the same accuracy score as using all data. Naive Bayes got tuned with the same values and the final accuracy ended on 83.53% and took 57 seconds to tune.

Logistic regression ended on the same, 90% and took 51 seconds to tune.

Random forest got 92.7 % accuracy after tuning, and this took 142 seconds to complete.

For the multilayer perceptron neural network I was able to tune it to 96.12% with a time on 53 seconds.

4 Discussion

From the obtained results I can conclude that the accuracy score is almost similar when comparing DataFrame models with the RDD models. The model that possibly in my opinion differs the most is the Random Forest. Though this mostly because of the technique/approach behind the model, it is picking random trees every time which means that each run of the model will not result in the same combination of trees every time. This also applies on the minimized version, the accuracy is very similar and random forest is the one that differs the most.

When measuring and comparing the time, we can see that the RDD solutions is take much more time to train, test and evaluate then the DataFrame does. This is probably because of the size of the data, when having big data the RDDs is supposed to perform worse in timing terms. The number of partitions that I tested is 2, 4, 6, 8, 15, 25 and 50. When comparing these results to the minimized version, in some models I was able to halve the computation time. Random Forest is similar, but for neural nets and Logistic regression, which took long time compute was almost halved.

There is still some improvements I want to obtain, such as the time it takes to train, test and evaluate the models. I can still see a big advantage of using spark instead of using one core libraries as sklearn. But I think with some more research about the clusters and workers Google provide, I may end up

with better models. By tuning the Random Forest model with more trees and deeper depth I was able to reach 97% accuracy but it took way to long for the model to be tuned and especially trained with the new parameters. Same for the Multilayer Perceptron Neural Network, I should be able to improve it to almost 99% but this requires more tuning which will increase the time and computations and therefore requires better hardware.

As a conclusion I can clearly see the advantage of using Spark when working with big data, and how much it can optimize performance. It is clear that for this specific project and dataset, DataFrame is the type that performs best.

5 Further Optimizations

Now based on the result that I got, there is probably things that can be optimized and that could possible improve the models. One optimization could be to try minimize the dimension of the data, for now each image consists of 784 pixels, which is a lot of features to process when training each model. This space could be reduced by using feature selection, where the selected features, will be the features that make the most affect on the model performance. There are some transformers within pyspark that can be imported from **pyspark.ml.feature** which can be used to select the best features. With a bit more research about each method I could possibly reduce the space and still achieve a pretty good accuracy. Another way to reduce the feature dimension could be to resize the image from 28x28 to 20x20 pixels, which means that I reduce the dimension almost by 400 features.

The last alternative for feature selection could be to build a K-Means model which will be used to select K features. The disadvantage of this model within this area is that I have to specify the number of features myself, and this can vary from 1 to 784 features. Another disadvantage of this model is that the initialization of the centroids which specifies the features is picked random.

In terms of accuracy I could try out more powerful hardware and extend the number of parameters that is tuned such that I can improve the accuracy for the models. Though this will probably increase the computation time of each model.

The last optimization for this project could be to study more in-depth about the memory in spark and try to come up with solutions that could possibly optimize the memory management.

References

[1] Jian Wu, Victor S Sheng, Jing Zhang, Hua Li, Tetiana Dadakova, Christine Leon Swisher, Zhiming Cui, and Pengpeng Zhao. Multi-label active learning algorithms for image classification: Overview and future promise. *ACM Computing Surveys (CSUR)*, 53(2):1–35, 2020.

[2] The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2021-05-20.

[3] Machine learning library (mllib) guide. https://spark.apache.org/docs/latest/ml-guide.html. Accessed: 2021-05-21.

[4] Mllib: Rdd-based api. https://spark.apache.org/docs/latest/mllib-guide.html. Accessed: 2021-05-21.