**PROJECT REPORT**

CMPE-256 - Large Scale Analytics



CIFAR-100 Image Classification using PySpark

Submitted By: Group 7

**Team Members:**

**Qiao Liu - 013802893**

**Ching-Min Hu - 013726154**

**Dandan Zhao - 013795392**

**Fernanda Bordin - 013800638**

**Megha Rajam Rao - 013709488**

**Rajasree Rajendran - 013774358**

**Google Colab link :** [**https://colab.research.google.com/drive/1y5jYOWa9KtmlcVe8gu3UY1IKYd0hAIzO?authuser=0#scrollTo=bUo18QAMPdL4**](https://colab.research.google.com/drive/1y5jYOWa9KtmlcVe8gu3UY1IKYd0hAIzO?authuser=0#scrollTo=bUo18QAMPdL4)

**TABLE OF CONTENTS**

[**1. Task Assignment**](#_heading=h.kbo3tvhuejs7) **4**

[2. Introduction](#_heading=h.2et92p0) **5**

[2.1 Libraries](#_heading=h.tyjcwt) 6

[2.2 Software & Tools](#_heading=h.3dy6vkm) 6

[3. Qualitative Comparison of PySpark MLib & Scikit-learn](#_heading=h.1t3h5sf) **7**

[4. Data preparation](#_heading=h.1y810tw) **9**

[**4.1 Extraction and pre-processing**](#_heading=h.u0dhhaokdunh) **9**

[**4.2 Data Validation**](#_heading=h.21fg85mla93i) **12**

[**4.3. Milestone 2 - One pair missing**](#_heading=h.o9gtbjycvl86) **15**

[5. Data Modeling](#_heading=h.1ci93xb) **16**

[**5.1 Extended EDA and Data transformation**](#_heading=h.vo3k7leaw361) **16**

[**5.1.1 Milestone 1**](#_heading=h.1u8tpzn8912q) **17**

[**5.1.2 Milestone 2**](#_heading=h.rcrwify4w9v4) **20**

[**5.2 Model Creation**](#_heading=h.lak9iusqa1cq) **21**

[**5.2.1 Milestone 1**](#_heading=h.3zx87ip4ix80) **21**

[**5.1.1.1 Logistic Regression**](#_heading=h.w28897k0ucbi) **21**

[**5.1.1.2 Naive Bayes**](#_heading=h.4arnrsvw1itq) **24**

[**5.1.1.3 Random Forest**](#_heading=h.qc1ldkeoni9e) **27**

[**5.2.2 Milestone 2 - We chose the best 2 models with superior accuracy based on Milestone 1.**](#_heading=h.mg791oqpnar7) **30**

[**5.2.2.1 Logistic Regression**](#_heading=h.3r5vykycwntg) **30**

[**5.2.2.2 Random Forest**](#_heading=h.xhsvqmj0nv48) **32**

[Code comparison](#_heading=h.z337ya) **45**

[**Results - with comparison**](#_heading=h.cmg025udv7o5) **56**

[**7.1 Logistic Regression**](#_heading=h.vy4o1luhfhfk) **56**

[**7.1.1 Milestone 1 - Prediction on Randomly Selected Testing Images Results**](#_heading=h.daq3hosy9luy) **56**

[Logistic Regression Confusion Matrix Comparison](#_heading=h.2bn6wsx) 57

[**7.1.2 Milestone 2 - Prediction on one testing subclass images from each of the two superclasses Results**](#_heading=h.sq290vo19fng) **57**

[**7.1.2.1 Scikit-Learn Results**](#_heading=h.p639zuqc8615) **57**

[**7.1.2.2 PySpark Results**](#_heading=h.l7nqwb3n220b) **60**

[**7.2 Random Forest**](#_heading=h.fnnk6u9uyz45) **61**

[**7.2.1 Milestone 1 - Prediction on Randomly Selected Testing Images Results**](#_heading=h.igx99kw96wyp) **61**

[**Random ForestClassifier Confusion Matrix Comparison**](#_heading=h.x3wrhjqlmx41) **61**

[**7.2.2 Milestone 2 - Prediction On One Testing Subclass Images From Each of the Two Superclasses Results**](#_heading=h.3hwzp6dtlujl) **62**

[**7.2.2.1 Scikit-Learn Results**](#_heading=h.ko632ssde3g) **62**

[**7.2.2.2 PySpark Results**](#_heading=h.11pezhaz58mn) **63**

[7.3 Quantitative comparison](#_heading=h.35nkun2) 64

[**Learning & Insights**](#_heading=h.k1jblihaj3dm) **67**

[9. Conclusion](#_heading=h.23ckvvd) **69**

[**Appendix:**](#_heading=h.p5n4hwu7hl8y) **71**

[**References**](#_heading=h.5ajld41sdta7) **71**

[**Referred Resources**](#_heading=h.8hk6s16n09z6) **71**

# 

# 

# 1. Task Assignment

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Description** | **Main Contributor** | **Other Contributor** |
| **Data Preparation** | Load, pre-process, validate and visualize data | Dandan Zhao,  Ching-Min Hu | Rajasree Rajendran,  Megha Rajam Rao, Fernanda Bordin, Qiao Liu |
| **ML methods** | Milestone 1 -  Logistic classification  Naive Bayes  Random Forest  Milestone 2 -  Logistic classification  Random Forest | Megha Rajam Rao,  Rajasree Rajendran,  Fernanda Bordin, Qiao Liu | Dandan Zhao,  Ching-Min Hu, |
| **PowerPoint presentation** | Overall  Input on Data preparation  Input on lessons learned | Fernanda Bordin | Rajasree Rajendran,  Megha Rajam Rao, Dandan Zhao,  Ching-Min Hu, Qiao Liu |
| **Project Report** | Overall  Input on Data Preparation  Input on graphics | Qiao Liu,  Rajasree Rajendran,  Megha Rajam Rao | Dandan Zhao,  Ching-Min Hu, Fernanda Bordin |

Table 1. Table of Task Assignment

**Selected ML Algorithm :** Naive Bayes (Milestone 1), Logistic Regression & Random Forest (Milestone 1 & 2)

# 2. Introduction

In this study, we compare the performance between PySpark MLib library and Scikit-learn using machine learning algorithms for image recognition. The chosen dataset is the widely known CIFAR-100. By training our models to recognize 2 classes of mammals: ‘medium-sized mammals’ and ‘small mammals’, we envision to discover the difference in the implementation of algorithms in PySpark MLib library and Scikit-learn.

The CIFAR datasets are labeled subsets of the 80 million tiny images dataset collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The images are of size 32x32 pixels with 3 color channels (RGB). It comprises of 100 classes containing 600 images each (500 training and 100 testing). The classes (fine labels) are grouped into 20 super classes (coarse labels) and corresponding classes. In this report we filtered the CIFAR-100 dataset to select images from the aforementioned super classes.

* Medium-sized mammals superclass includes the following 5 subclasses:

**fox, porcupine, possum, raccoon, skunk.**

* Small mammals superclass includes the following 5 subclasses:

**Hamster, mouse, rabbit, shrew, squirrel.**

The models were designed to determine the superclass group the mammal belongs to. Further, in milestone 1, we performed a random split of the dataset into 80% training set and 20% testing test. Milestone 2 was an elaborate study of the pairwise results. Testing set included a single pair of small mammal & large mammal, and training set included the remaining subclasses. As there are 5 subclasses in each superclass, we generated 25 pairs and compared the results of their classification of images into superclasses. A salient feature is the comparison of the methodology and results of the models using Pyspark and Scikit-learn. In this subsequent sections, we will delve into details and juxtapose the 2 approaches based on accuracy and speed. We will also discuss our personal experience on working with the 2 different approaches.

## 2.1 Libraries

1. NumPy
2. Scikit-learn
3. Matplotlib
4. Math
5. Seaborn
6. Spark MLib library

## 2.2 Software & Tools

1. Google Colaboratory
2. Python (language)
3. Microsoft PowerPoint (presentation)
4. Microsoft Word (report)
5. Google drive (document sharing)

# 3. Qualitative Comparison of PySpark MLib & Scikit-learn

Ever since the advent of the computational era, technologies have been juxtaposed and compared. We performed a quick survey that revealed the popular opinions of academics and data practitioners from the industry. According to Villu Ruusmann, distributed systems such as Spark work best when they implement simple models for large datasets. Figure 3.1 sheds light on the difference in performance of serial, parallel and distributed systems. Although more complex from an algorithmic perspective, Serial models have witnessed diminishing performance with large datasets. Parallel model fair better, whereas distributed systems are the frontrunners with superior performance. They are designed and equipped to handle extremely large datasets

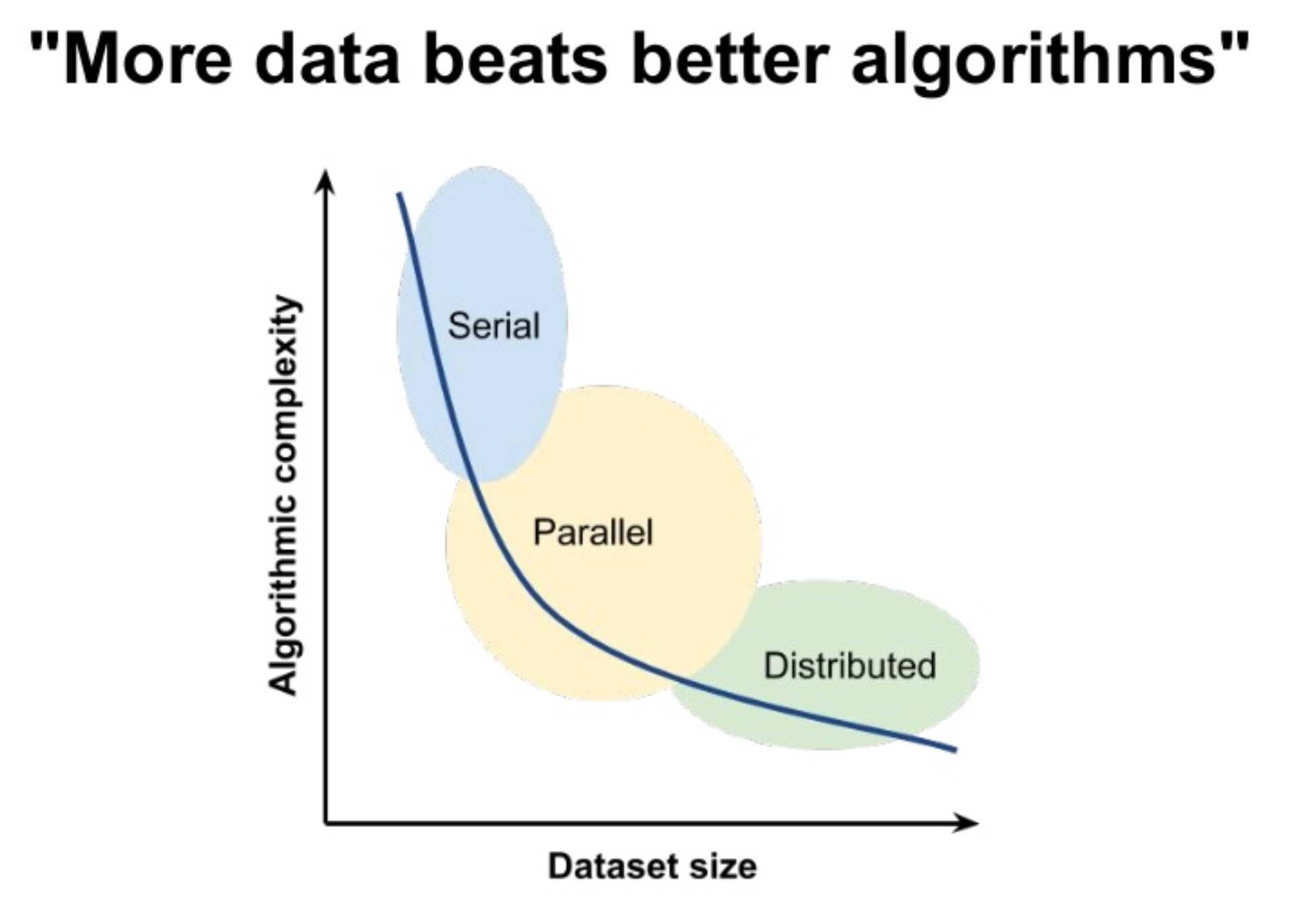


Fig 3.1: Dataset size and algorithmic performance

Yet Scikit-learn, which happens to be popular and widely adopted, thrives due to its streamlined functions, ease of use, optimization tools, rich ecosystem and the advantage of better visualization tools and rich ecosystem. Below tables compare the pros and cons of the Scikit-learn and PySpark.

**Table of Comparison - Advantages of Machine Learning (ML) with Scikit-learn & PySpark**

|  |  |
| --- | --- |
| **ML with Sckit-learn** | **ML with Spark (Python)** |
| Easy to use, especially streamlined functions. | Uses caching to reuse data |
| Great visualization tools (Pandas and Matplotlib) | Has accumulators (keep state across iterations) |
| Rich ecosystem (many libraries) | Fault tolerance |
| ML models run smoothly and are easy to optimize | Popular algorithms supported |

*Table 3.1 Table of Comparison between the advantages of Machine Learning (ML) with Scikit-learn and ML with Spark in Python environment*

**Table of Comparison - Disadvantages of Machine Learning (ML) with Scikit-learn & PySpark**

|  |  |
| --- | --- |
| **ML with Sckit-learn** | **ML with Spark (Python)** |
| Limited to one machine | Takes a long time to aggregate dataframe (ML Lib not efficient) |
| Not advisable for extremely large datasets | Memory expensive |
| High latency |
|  | Handy simple functions missing for tasks such as classification matrix or printing null values. |

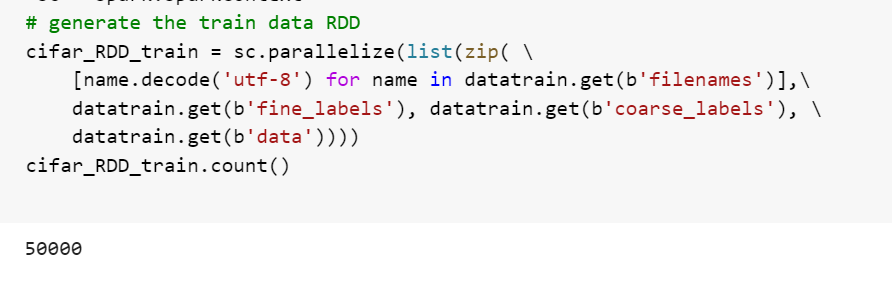
Table 3.2 Table of Comparison between the disadvantages of Machine Learning (ML) with Scikit-learn and ML with Spark in Python environment

# 4. Data preparation

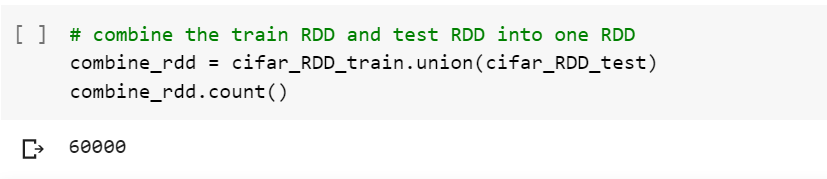
# 

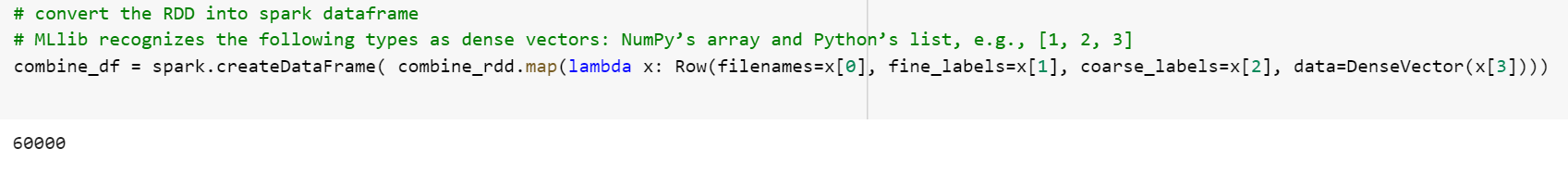
# 4.1 Extraction and pre-processing

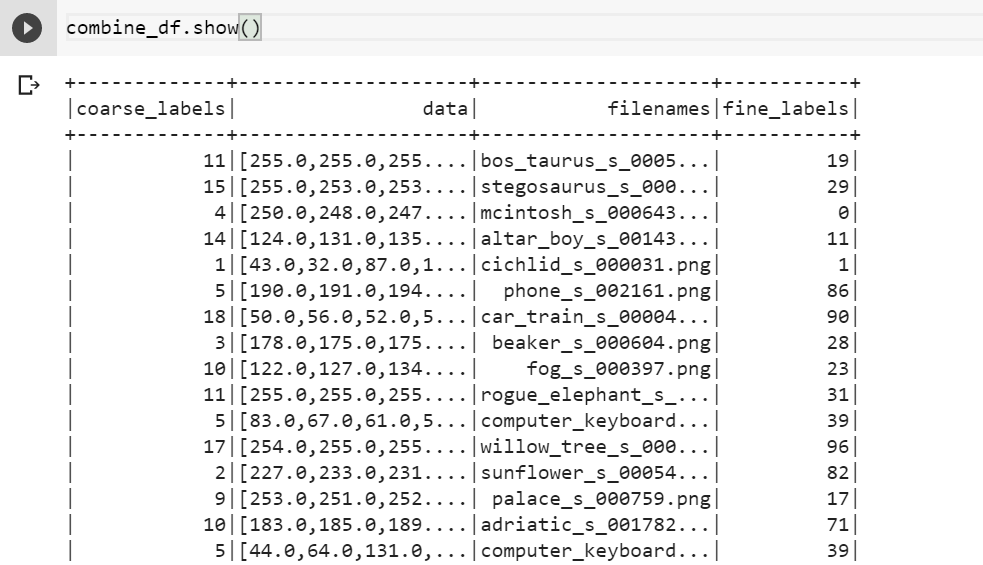
* In order to download data, we installed OpenJDK and findspark packages to set up the environment and build Spark session. Then, we imported all the necessary packages and modules such as Spark SQL and MLib.
* Reading in the data with PySpark is not as straightforward as importing from Keras datasets with ***cifar100.load\_data()*** function. With much effort and experimentation, we downloaded the python version of CIFAR100 dataset from online source: <https://www.cs.toronto.edu/~kriz/cifar.html>.
* The downloaded dataset includes three files: training data, test data and metadata. We converted the training and test data into Spark RDD.
* Then we applied the ***unpickle*** function provided by <https://www.cs.toronto.edu/~kriz/cifar.html> to convert the dataset into dictionaries. We chose four key-value pairs: {filename: name of the image}, {coarse label: index of superclass}, {fine label: index of fine class}, and {image data: list of pixels of the image} to generate the RDD for later operation. The values of filename consisted of the name of the image, the type was byte, and we had to convert the byte to string type.
* With ***zip*** function, we transformed values with different keys into one tuple for each record, then used ***parallelize*** function to generate the RDD for both train and test data.



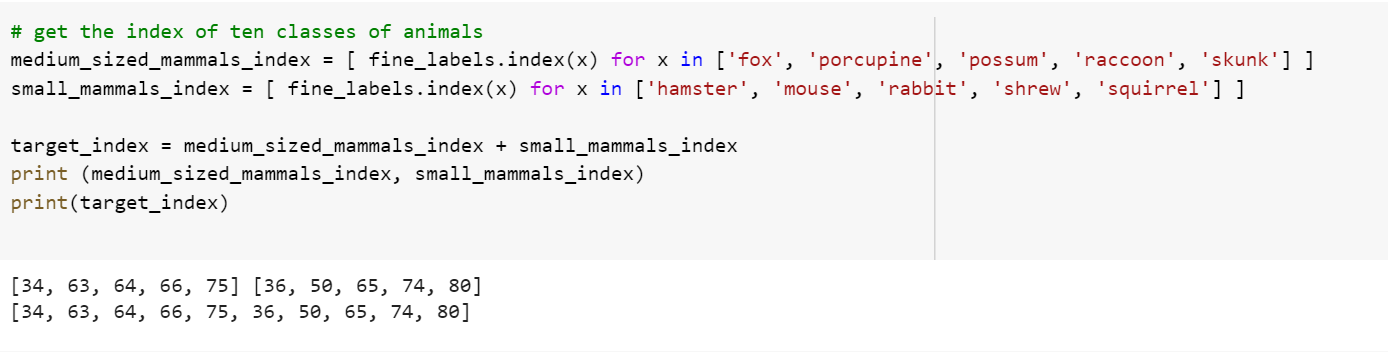
* With ***union*** function, we combined test RDD and train RDD. The total count of records is 60000.



* We transformed the RDD into Spark DataFrame by with ***createDataFrame()*** and ***map()*** function. During this step, we converted the data type to ***DenseVector*** so Spark can recognize them as arrays of value.
* In our DataFrame, there are four columns, ***filename, coarse label, fine label*** *and* ***data***.



* We generated indexes of 2 superclasses as a list. The assigned superclasses(small mammals and medium-sized mammals) were filtered by using given metadata.



* We filtered out the assigned classes with ***filter..isin()*** function.



## 

## 

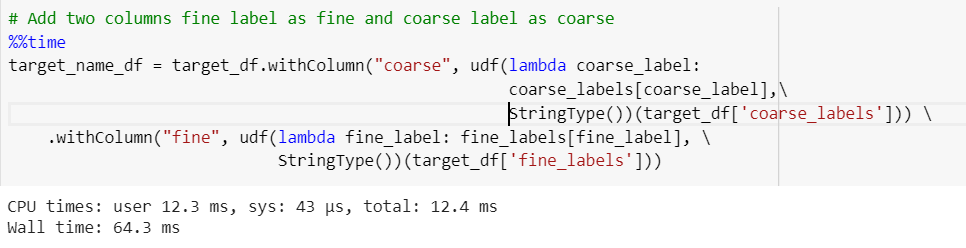
## 

## 

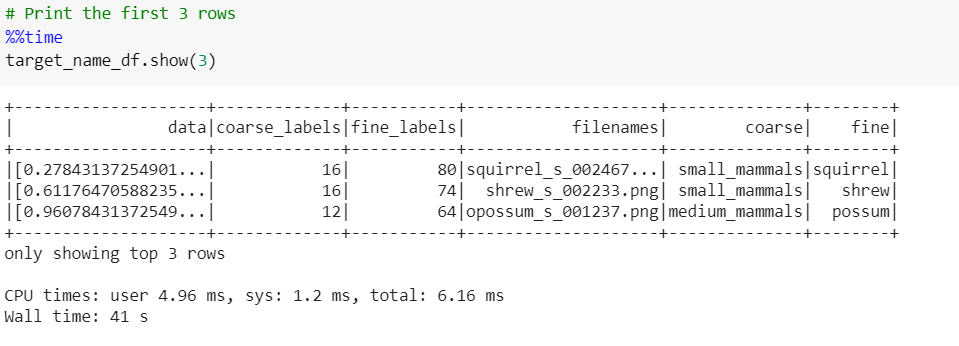
## 

## 4.2 Data Validation

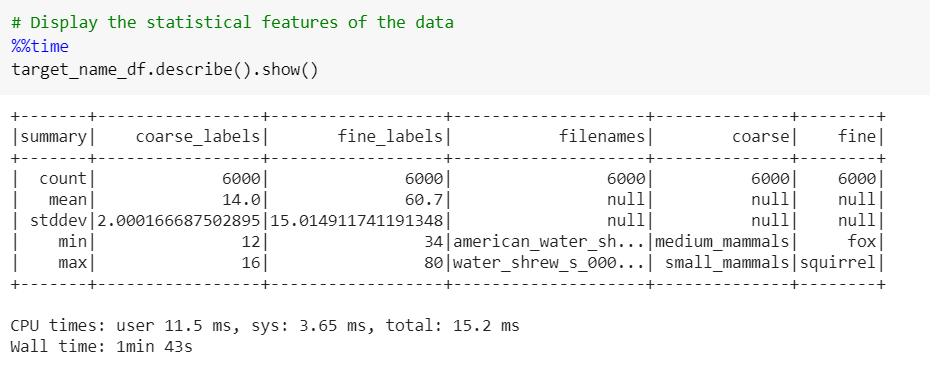
* We validated our data on the generated DataFrame using ***.withColumn()*** function, we added two columns ***coarse*** and ***fine*** which contain the verbal labels for each record.



* Let’s check if the columns were created correctly.

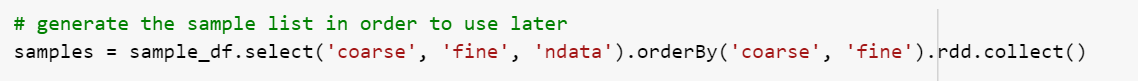


* Check statistical features of the data.

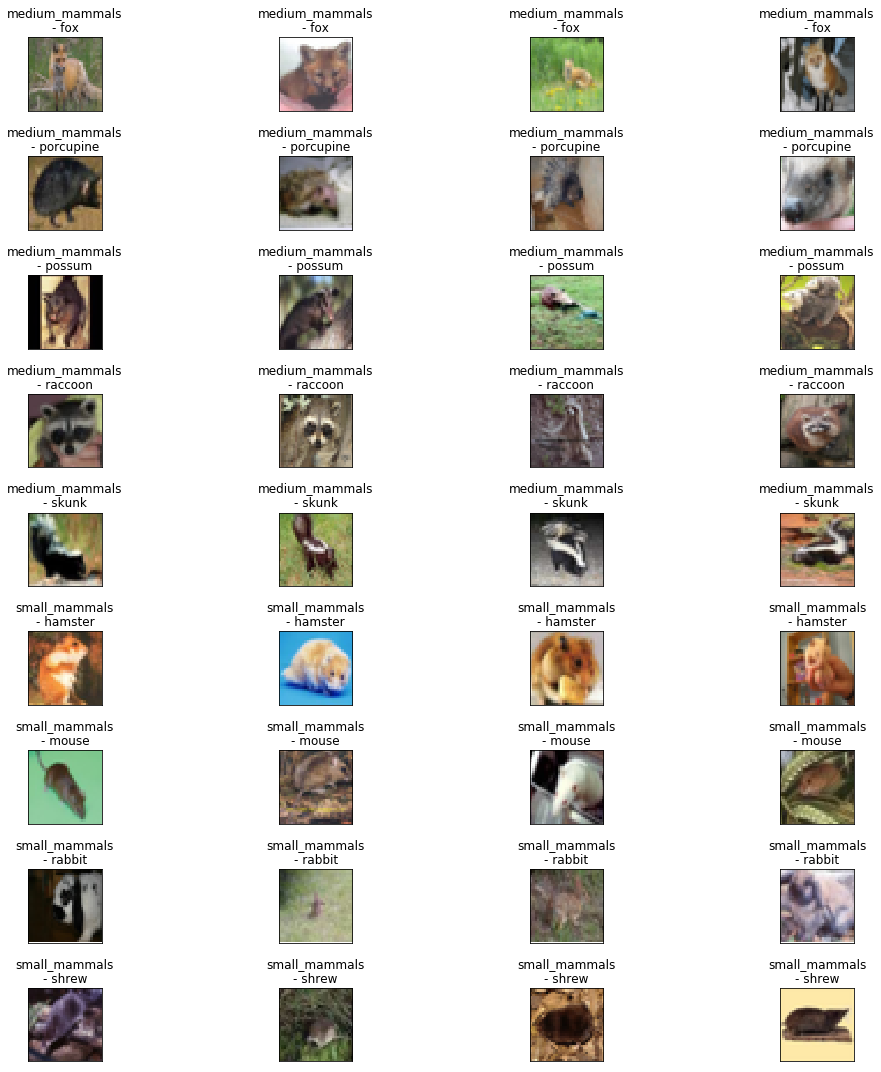
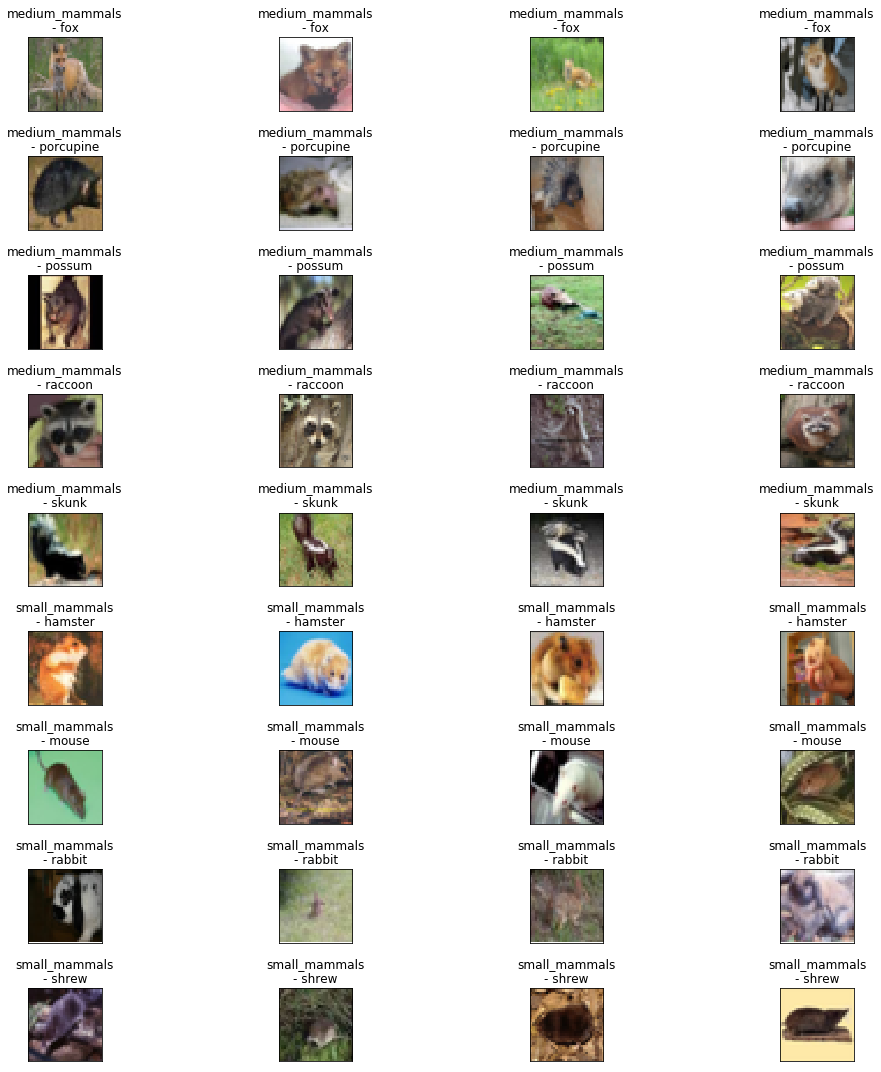


* We used ***sampleBy()*** function to generate a sample DataFrame for validation purpose and transformed it back to RDD with the ***.collect().*** We got the list containing all records including data and labels, then randomly picked 4 samples for each class to perform visual inspection and validation.





* Visualization was an essential step towards validation of the filtered data. We generated the 4 random images from each subclass using ***matplotlib***. As instructed, the title included the verbal superclass name(coarse label) and subclass name (fine label). The following were the validation images, we manually check the images, they all matched their fine labels and coarse labels.

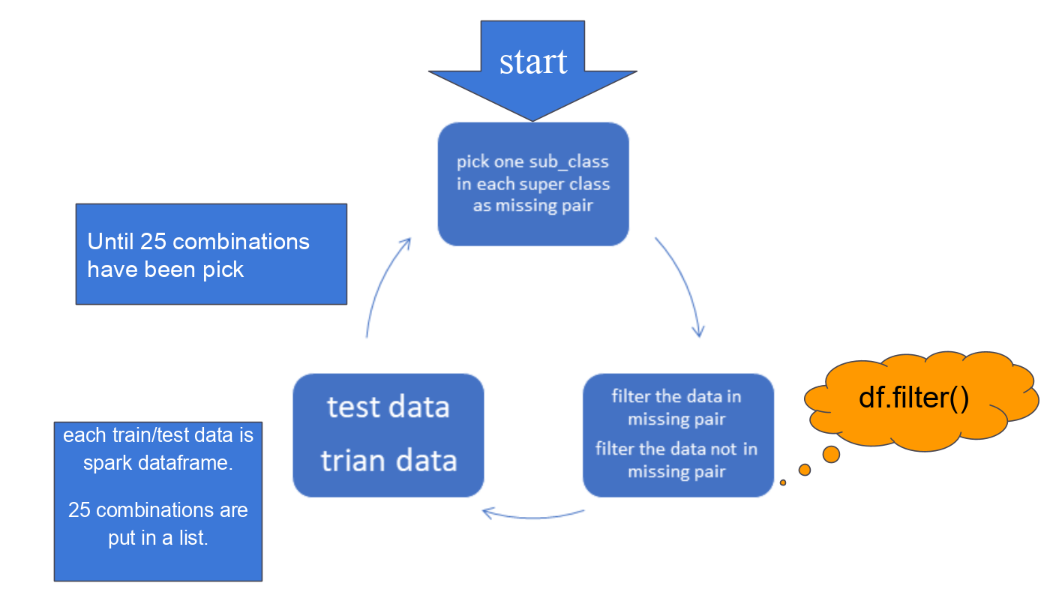


* Once the dataset was validated and ready, we proceeded to model training.

## 

## 4.3. Milestone 2 - One pair missing

In the pipeline of one missing pair, each one is an RDD dataframe. First, need to pick the missing one subclass (fine\_label) in each superclass (coarse\_label) as a missing pair, then use ***.filter()*** to filter out the whole dataframe with fine\_label matched the selected labels as test data and the fine\_label does not matched as train data. This process would execute for 25 times until all combinations have been done. Each test/train data was appended in a list.



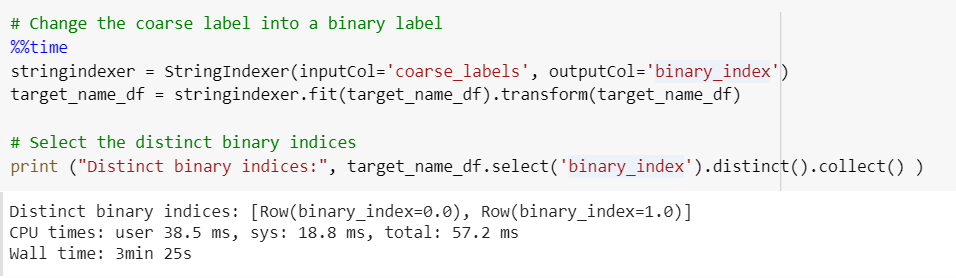


# 5. Data Modeling

## 5.1 Extended EDA and Data transformation

Although we did substantial pre-processing in the Data preparation stage, PySpark algorithms failed to process the features and target value. These algorithms have a specific format for data ingestion. Hence, we embarked on an elaborate exploratory data analysis (EDA) to unveil the features of the dataset and transform into an acceptable format. Below are the screenshots and steps followed.

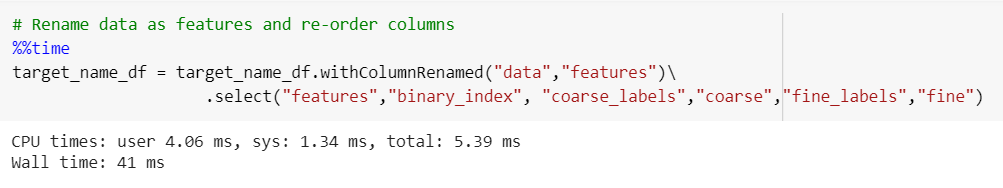
* As we are defining our binary classification problem, we first convert and combine the coarse labels of each superclass into a binary label using ***StringIndexer*** function.



* Data integrity was ensured by checking null values in the dataset, which were absent.

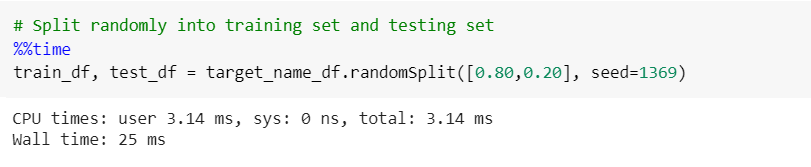


* The input feature, with the image data, was renamed to ‘features’ and relevant columns were selected and re-arranged.

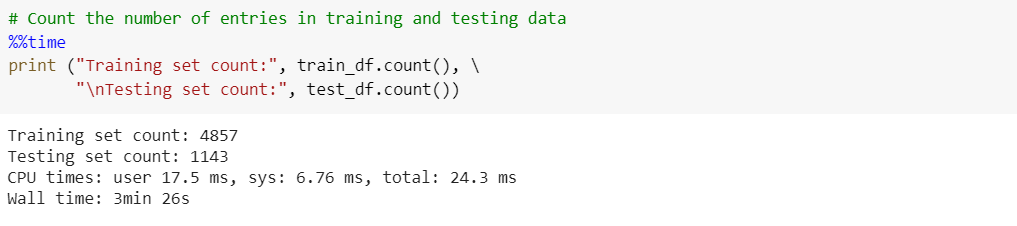


### 5.1.1 Milestone 1

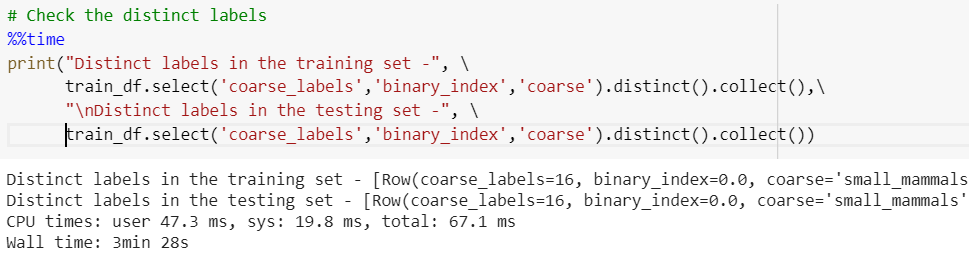
* The dataset was split into training and testing sets using ***randomSplit*** function. Here, we randomly select 80% of data as training data and remaining 20% data as testing data. Seed is used to save the state of the random function in subsequent executions.



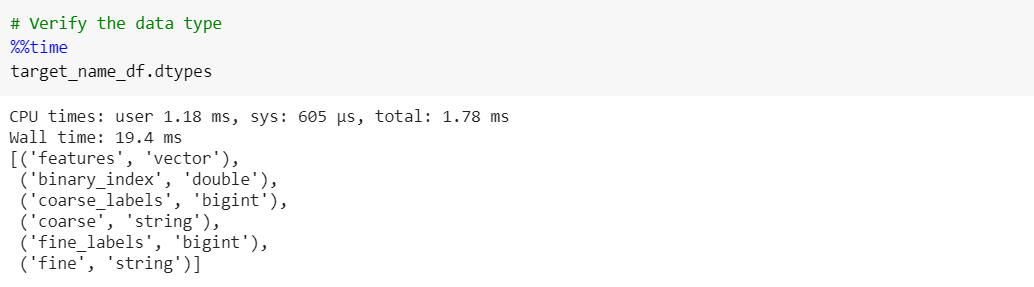
* Training and testing counts were checked using ***.count()*** function.



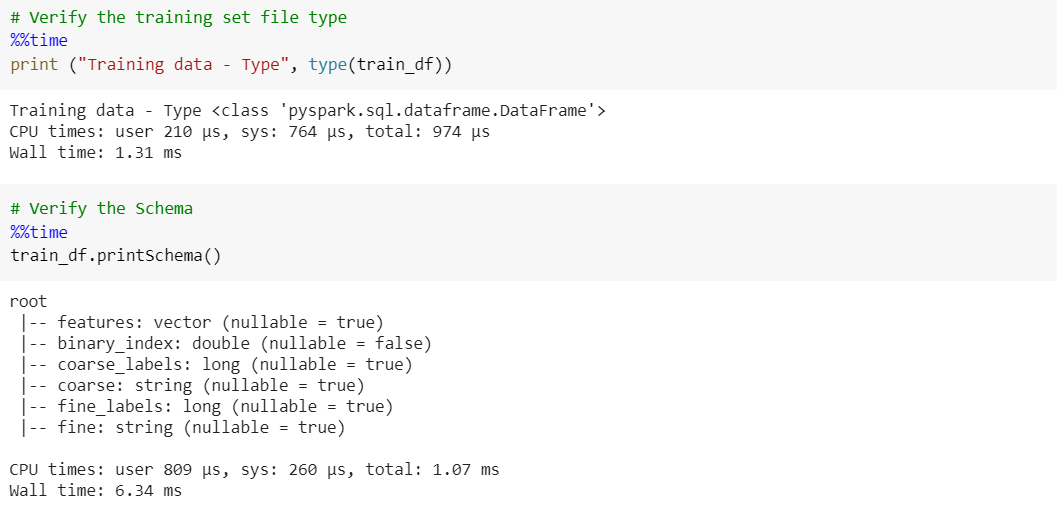
* Distinct superclass labels were checked by using ***.distinct()***function.



* Datatypes of the final file was checked to ensure adherence with the specified data format, which is vector image data and binary labels.

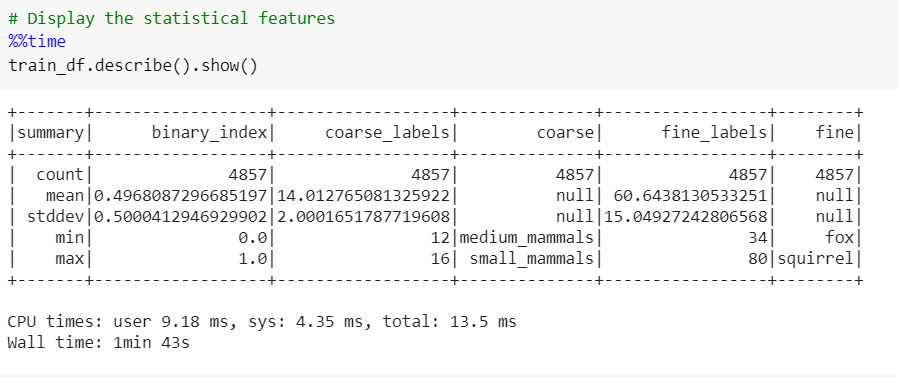


* Further, we performed the below steps to deeply understand the training and testing data.

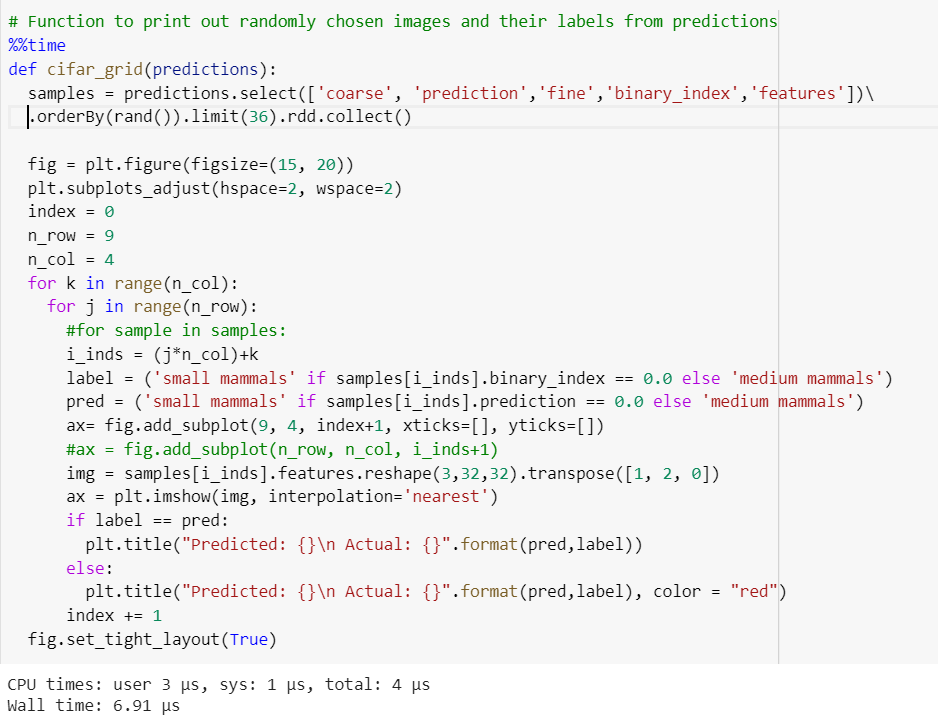


We checked the type of file and verified the schema using ***type()*** and ***printSchema().***

* The statistical features were checked using ***.describe().show().***

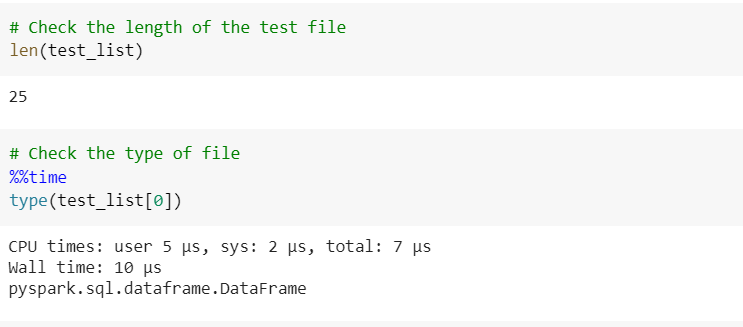


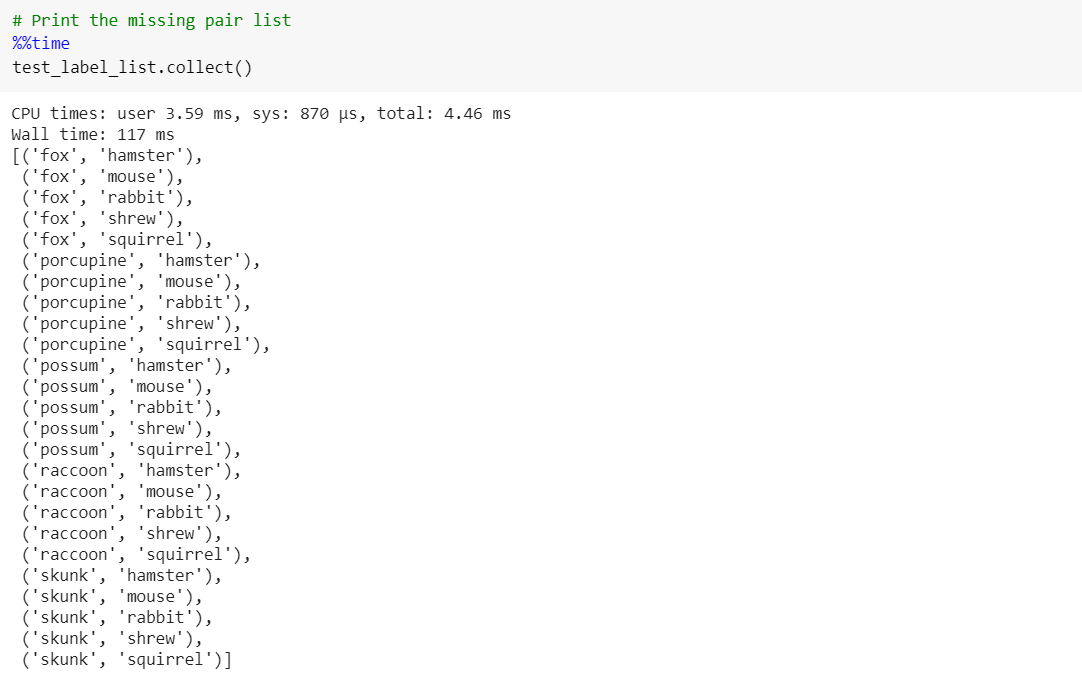
* Create a function for displaying the expected and predicted result.



### 5.1.2 Milestone 2

* EDA and data transformation was similar. Additionally, we checked the length and type of the data. There were 25 pairs were found in PySpark Dataframes.



* The test labels were checked and verified.

## 5.2 Model Creation

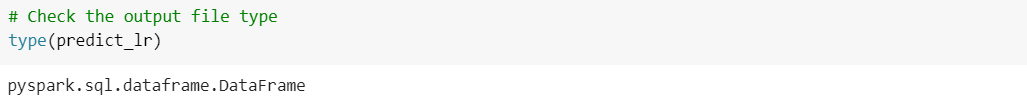
### 5.2.1 Milestone 1

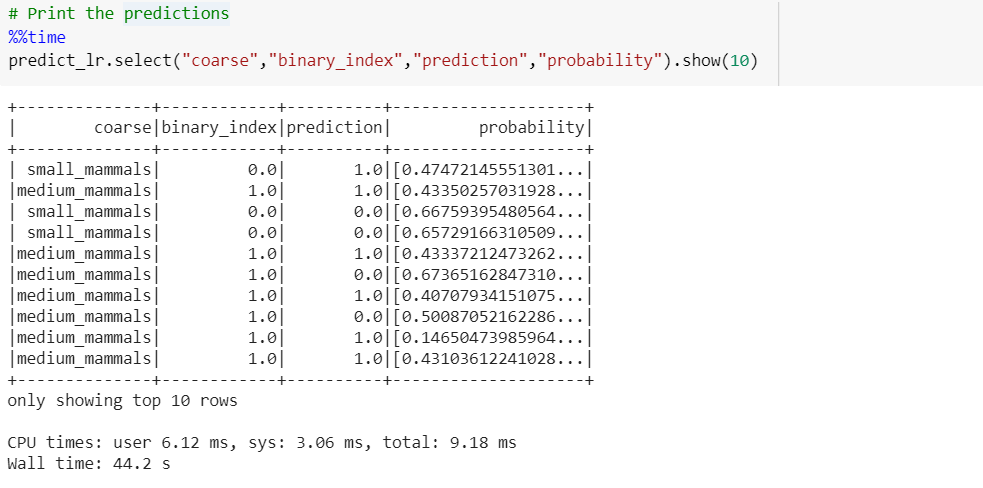
#### 5.1.1.1 Logistic Regression

* Using***LogisticRegression()*** function from pyspark ml library, we instantiate the model and fit the same using training data. Thereafter, we predict by transforming the testing data, and calculate the score using ***BinaryClassificationEvaluator()*** function.

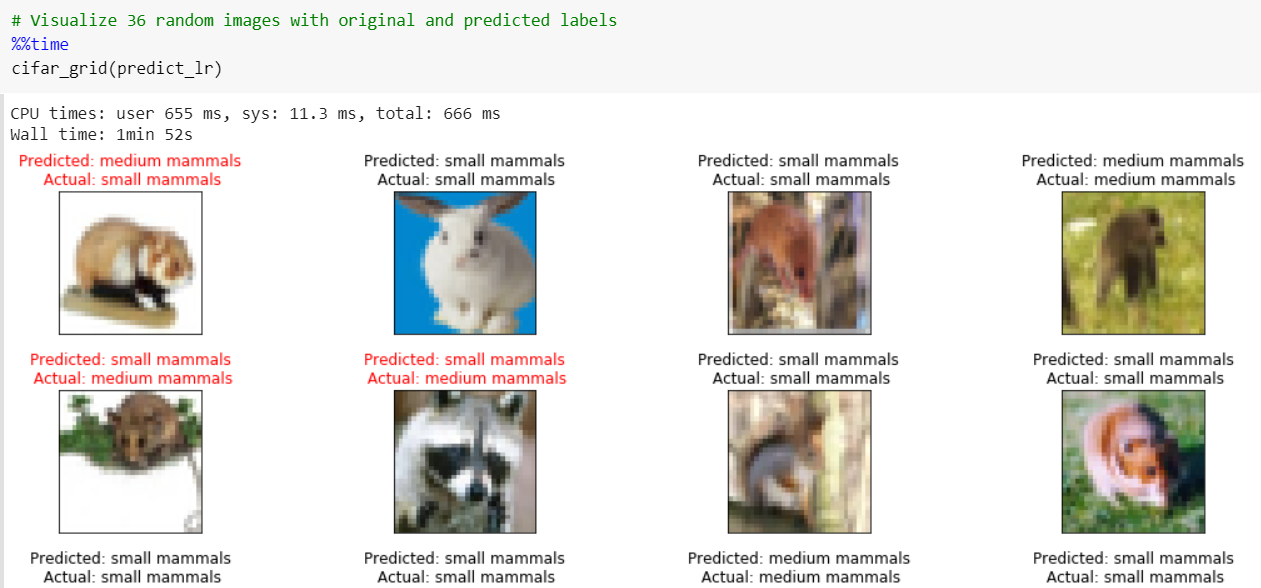


* The results are printed in the subsequent step. The output file is a Pyspark dataframe.





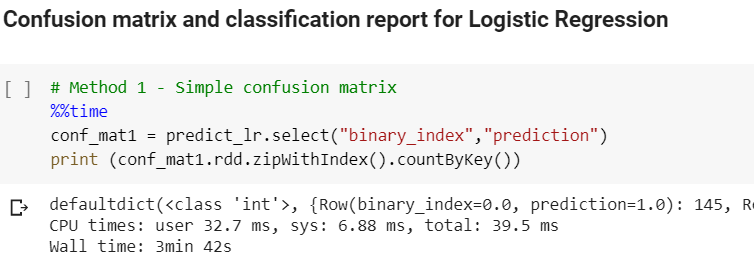
* 36 random images with the expected and predicted results were displayed using the aforementioned function.

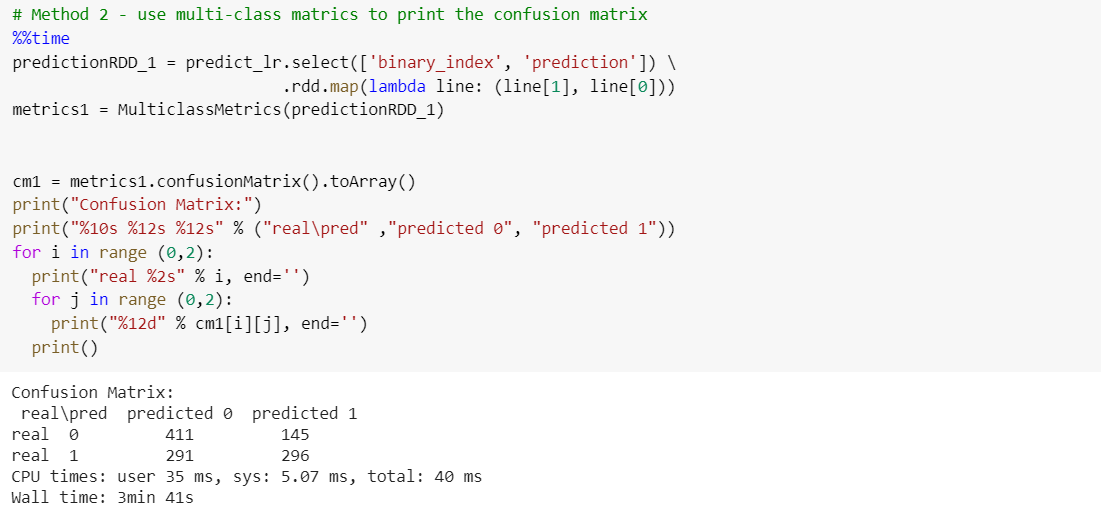


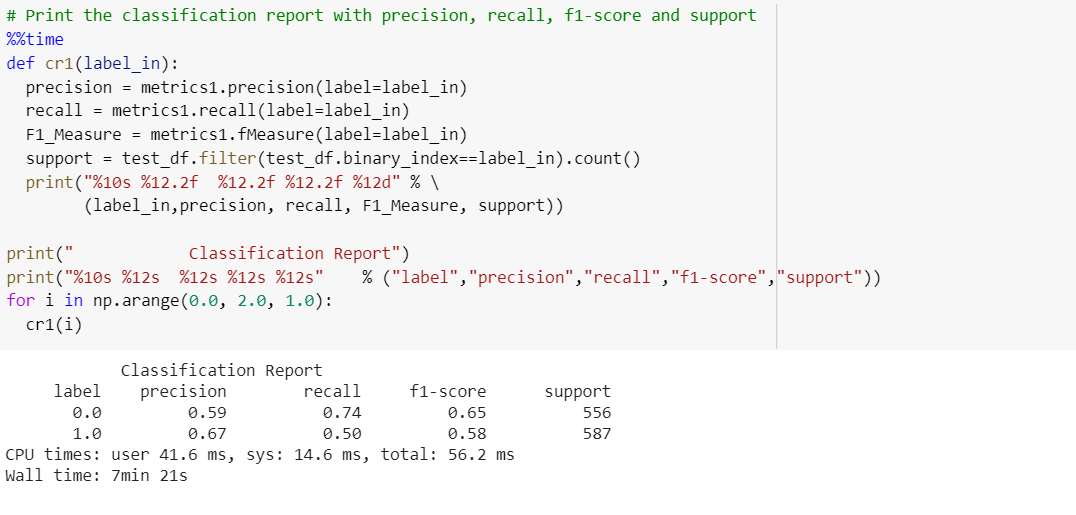
* Confusion matrix and classification report printing was a more elaborate task as PySpark lacks handy functions.

**Note:** Comparison with Scikit-learn is provided in later sections.

We used 2 different methods to print the confusion matrix. ***zipWithIndex()*** function enabled us to add index to the RDD. ***MulticlassMetrics()*** function was used in the second method.





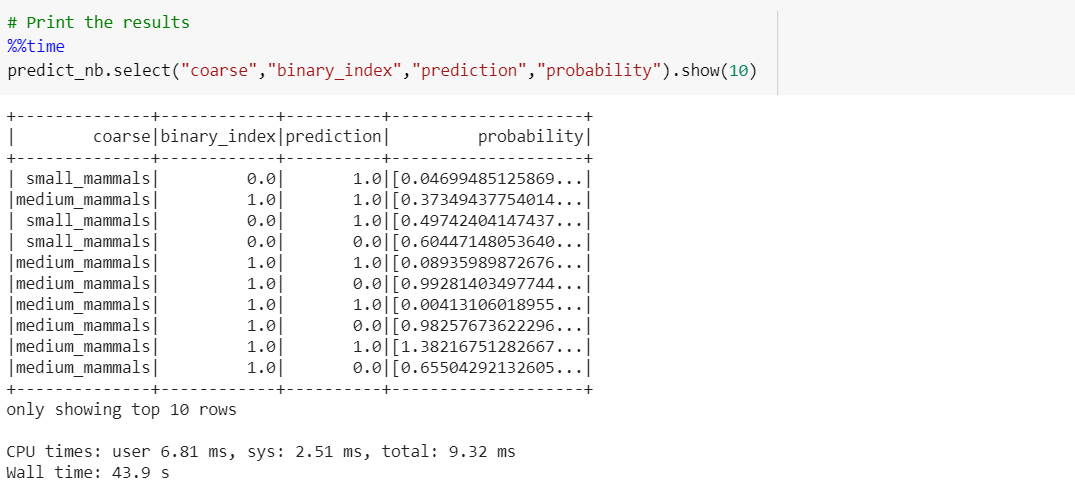


#### 5.1.1.2 Naive Bayes

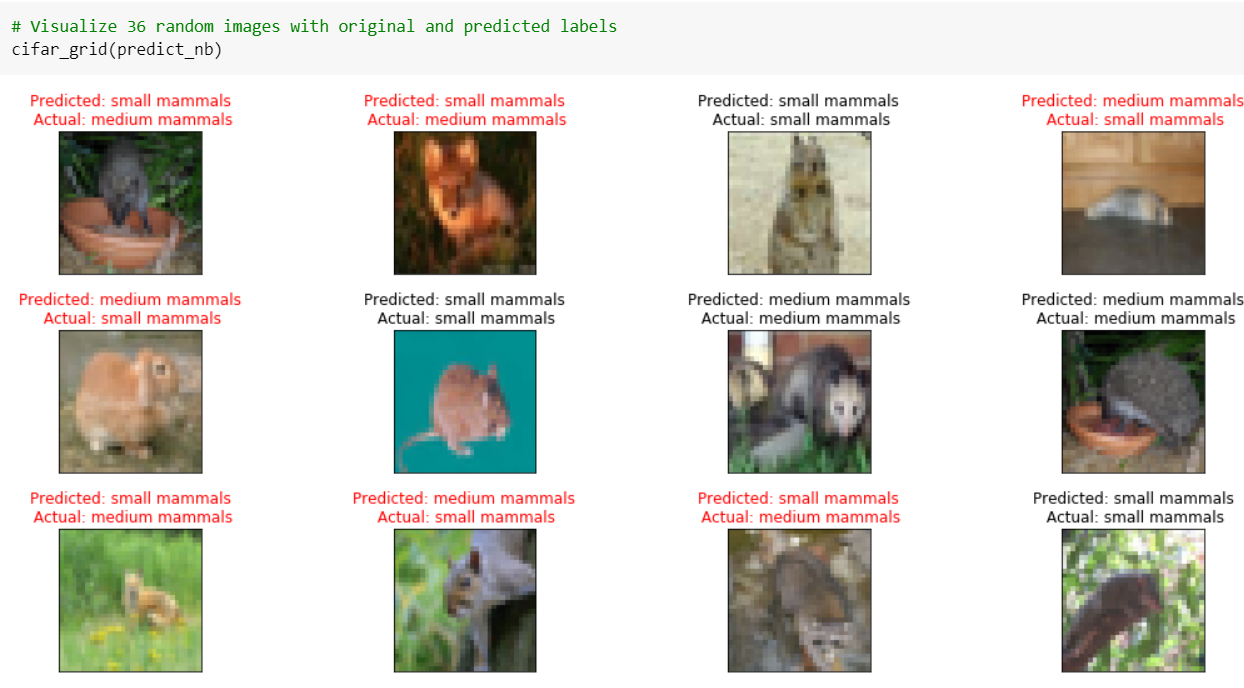
* Using***NaiveBayes()*** function, we instantiate the model and fit the same using training data. Thereafter, we predict by transforming the testing data, and calculate the score using ***BinaryClassificationEvaluator()*** function.



* The results are printed in the subsequent step. The output file is a Pyspark dataframe.



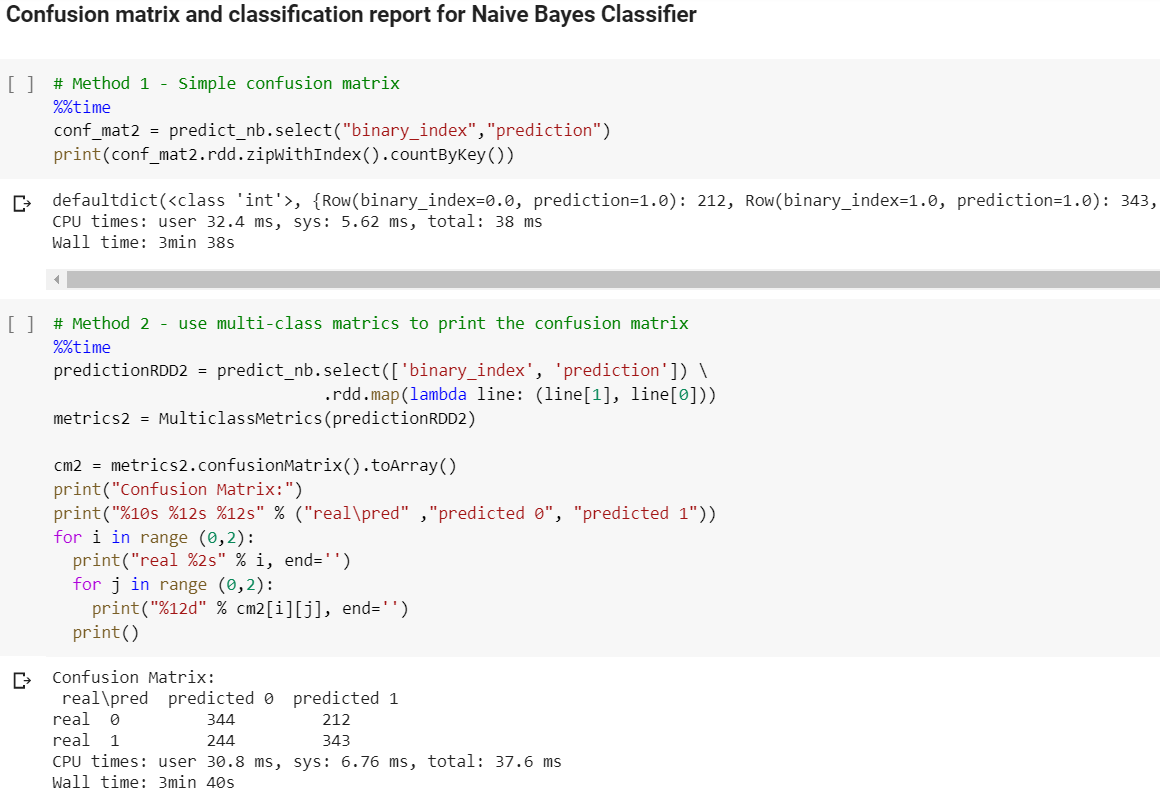
* 36 random images with the expected and predicted results were displayed using the aforementioned function.



* Confusion matrix and classification report printing was a more elaborate task as PySpark lacks handy functions.

**Note:** Comparison with Scikit-learn is provided in later sections.

We used 2 different methods to print the confusion matrix. ***zipWithIndex()*** function enabled us to add index to the RDD. ***MulticlassMetrics()*** function was used in the second method.





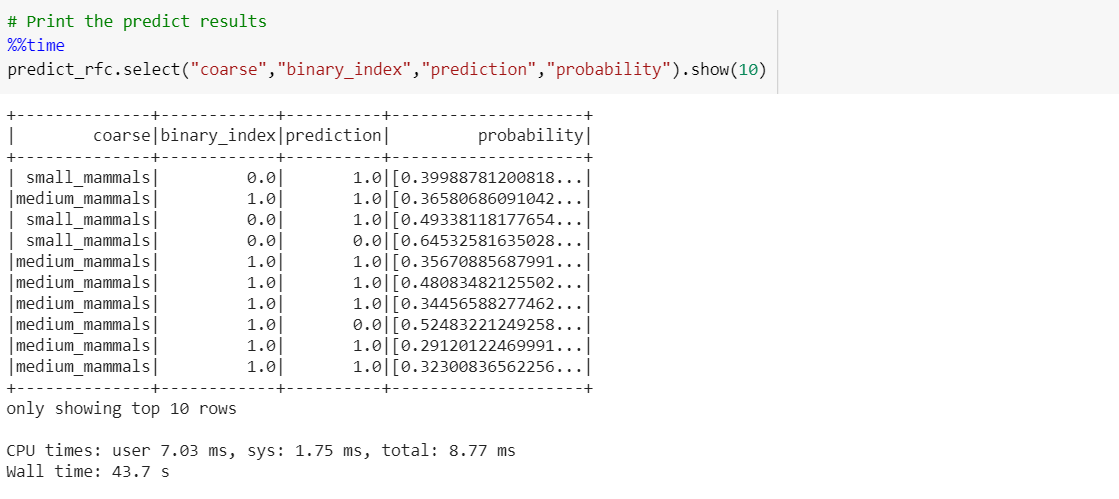
Please refer subsequent sections for comparison with Scikit-learn.

#### 5.1.1.3 Random Forest

* Using***RandomForestClassifier()*** function, we instantiate the model and fit the same using training data. Thereafter, we predict by transforming the testing data, and calculate the score using ***BinaryClassificationEvaluator()*** function.



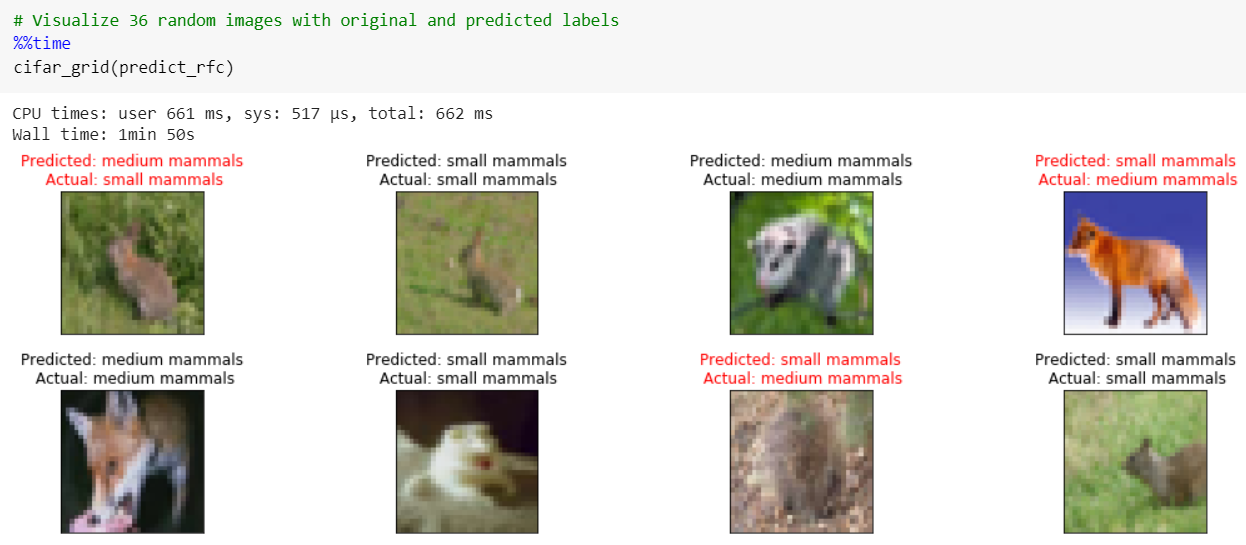
* The results are printed in the subsequent step. The output file is a Pyspark dataframe.



* The parameter grid was checked to study the default values. For example, numTrees is 20 and maxDepth is 5 by default.



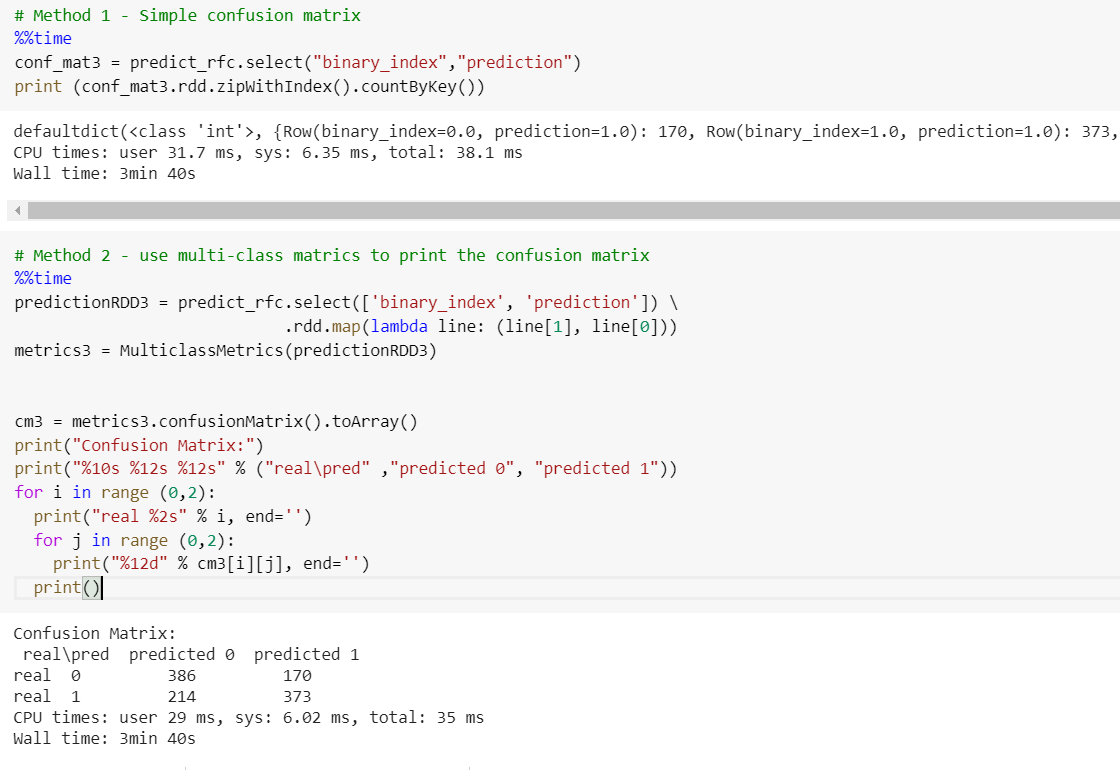
* 36 random images with the expected and predicted results were displayed using the aforementioned function.

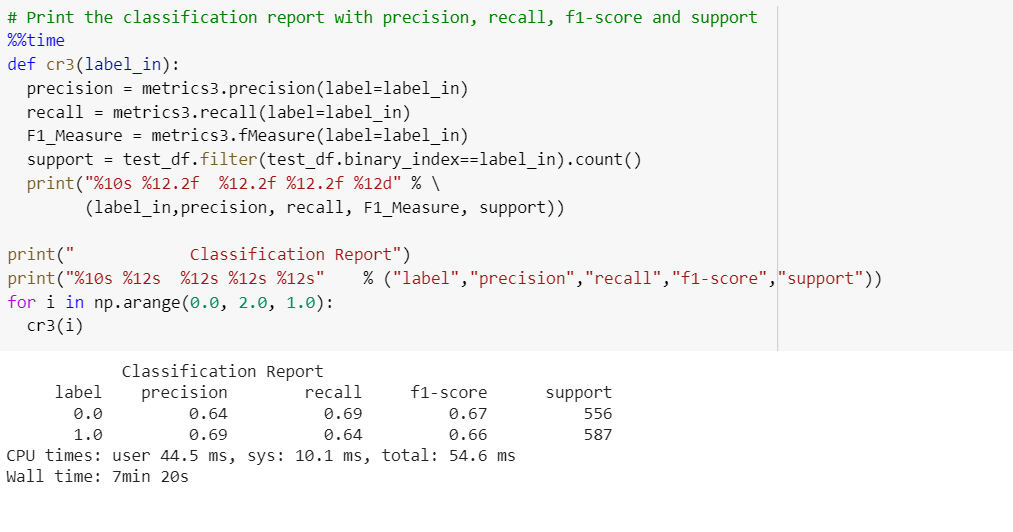


* Confusion matrix and classification report printing was a more elaborate task as PySpark lacks handy functions.

**Note:** Comparison with Scikit-learn is provided in later sections.

We used 2 different methods to print the confusion matrix. ***zipWithIndex()*** function enabled us to add index to the RDD. ***MulticlassMetrics()*** function was used in the second method.



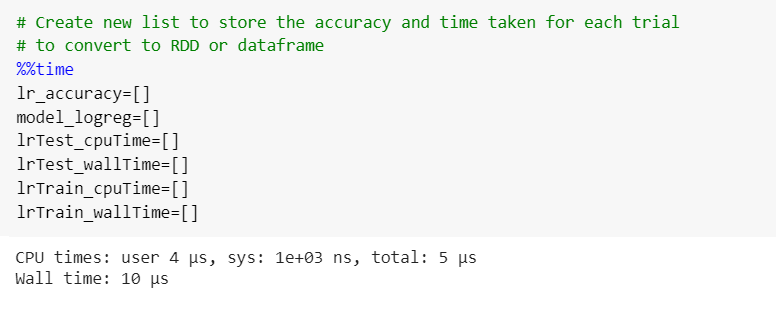


Please refer the subsequent sections for comparison with Scikit-learn.

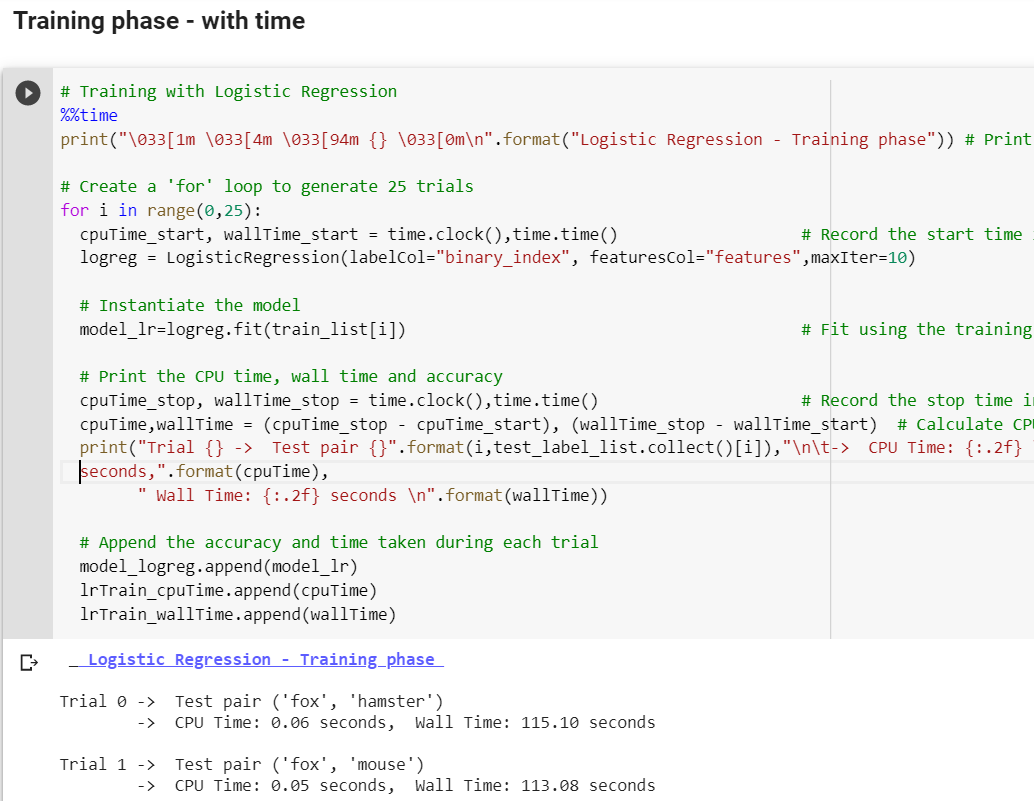
### **5.2.2 Milestone 2 -** We chose the best 2 models with superior accuracy based on Milestone 1.

#### **5.2.2.1 Logistic Regression**

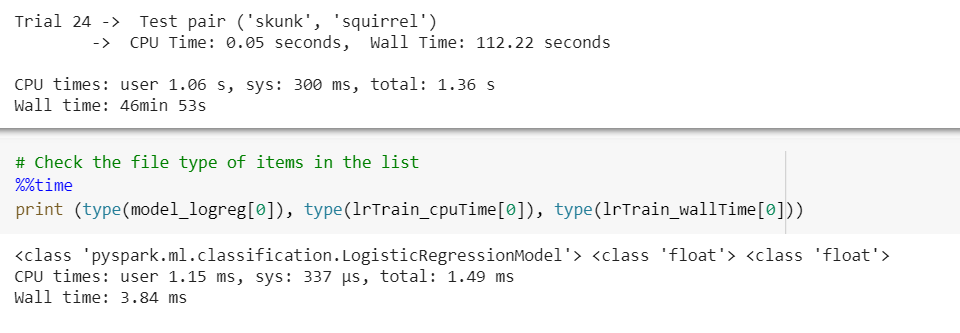
* Empty lists were initialized to store the model inputs. They were thereafter converted to Pyspark dataframes.



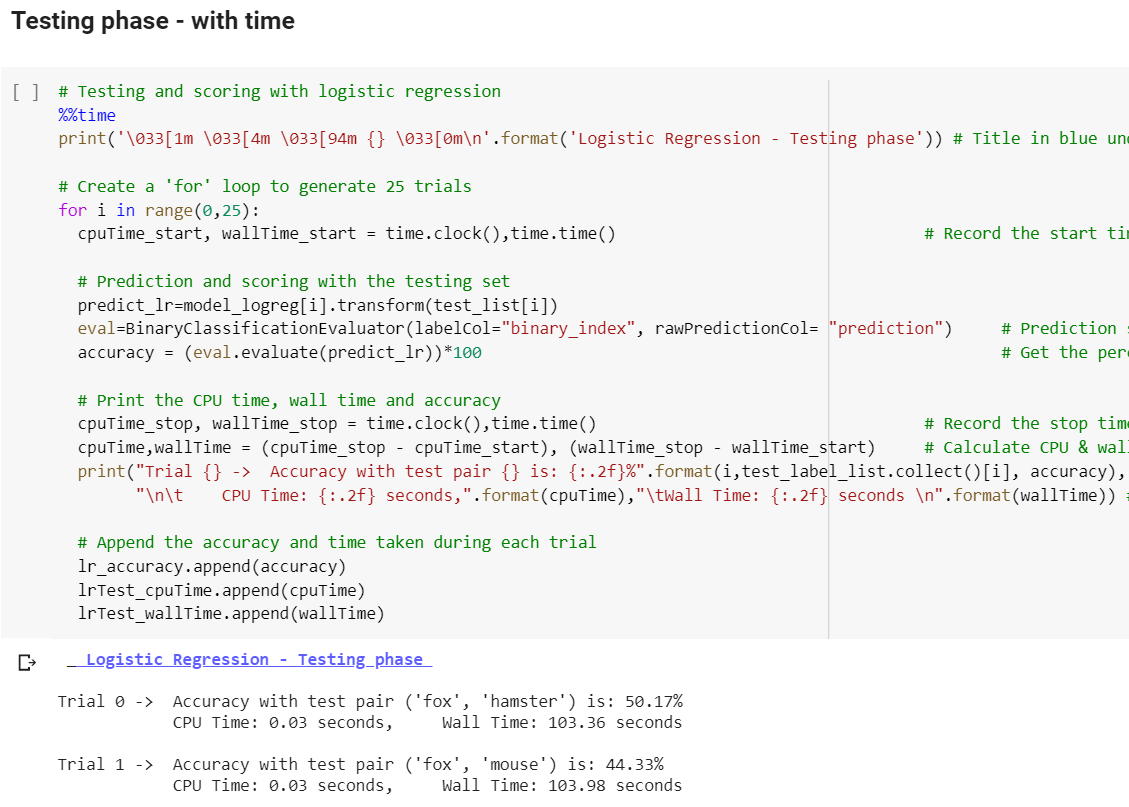
* Training was performed using the same function from milestone 1. However, we recorded the time using ***time.clock()*** for CPU time and ***time.time()*** for Wall time. We had to create a for loop to execute the training for 25 pairs and store results in a list.

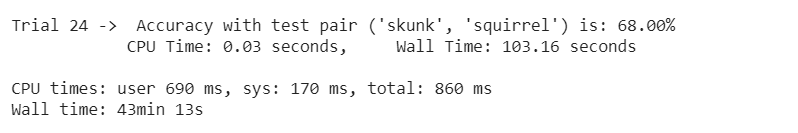


Although magic operator ***%%time*** gave the final time taken, our calculated value is lesser as it excludes loops, print statements and append commands.



* Similarly, testing was performed.

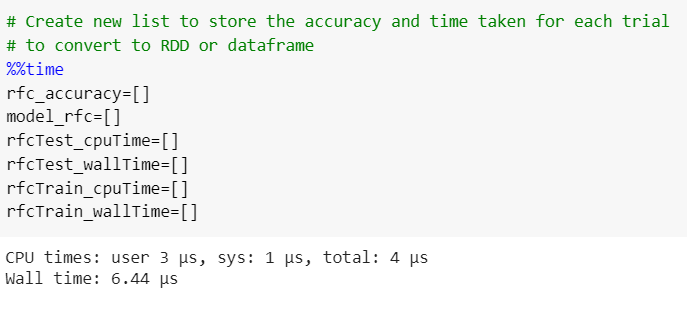




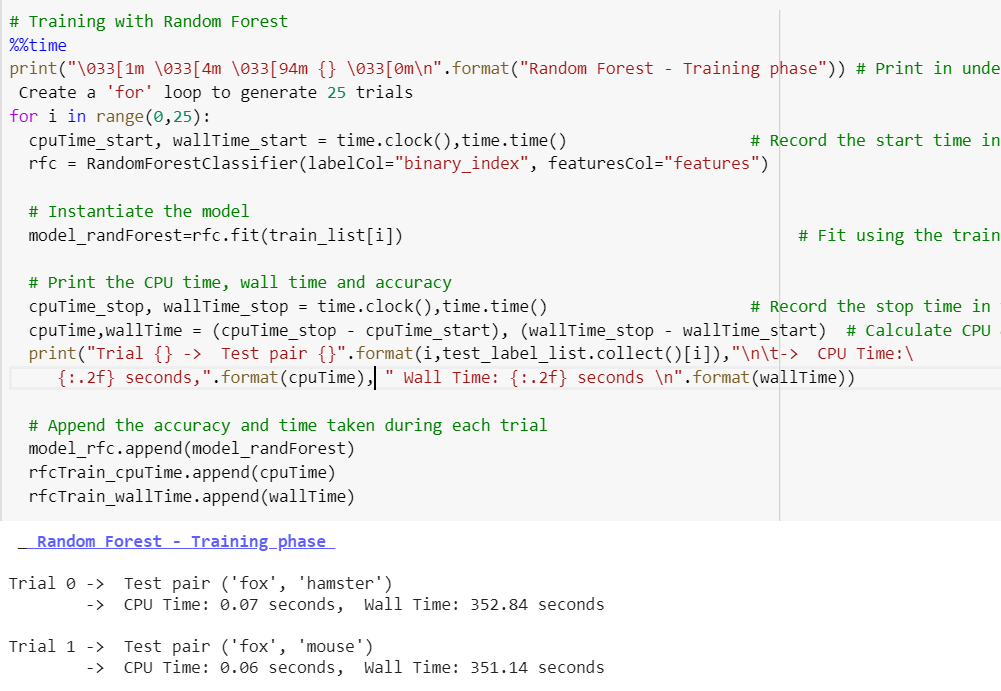
Please refer the subsequent sections for comparison with Scikit-learn.

#### **5.2.2.2 Random Forest**

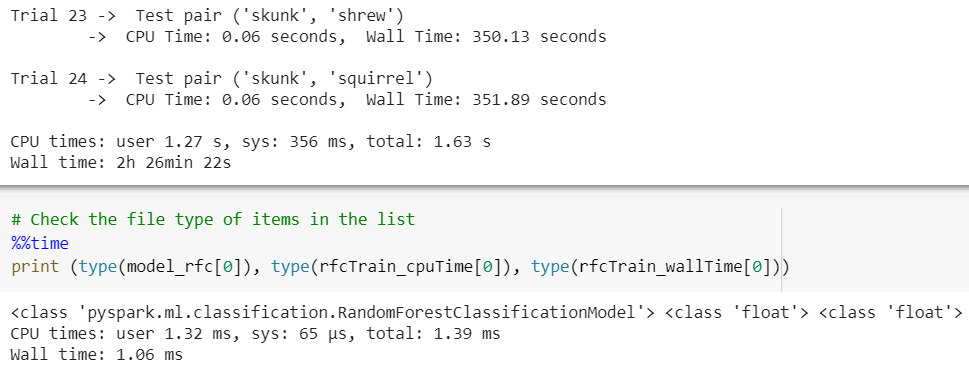
* Empty lists were initialized to store the model inputs. They were thereafter converted to Pyspark dataframes, once the lists were populated.



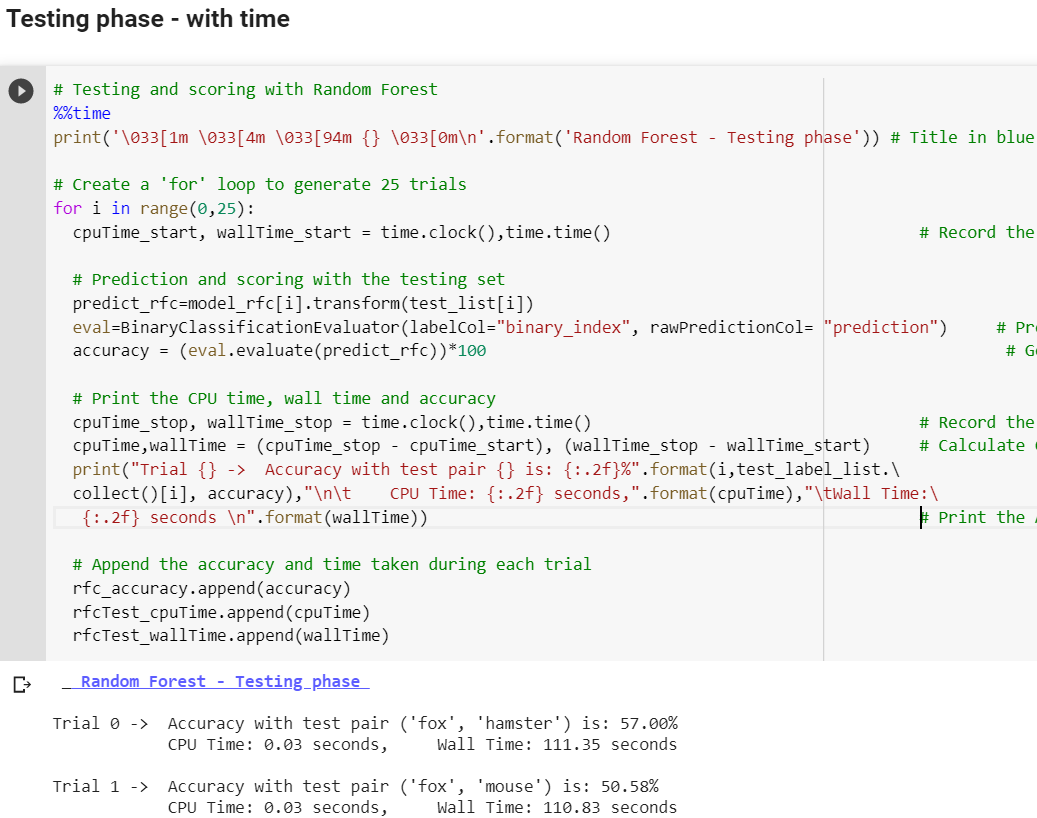
* Training was performed using the same function from milestone 1.
* However, we recorded the time using ***time.clock()*** for CPU time and ***time.time()*** for Wall time. We had to create a for loop to execute the training for 25 pairs and store results in a list.

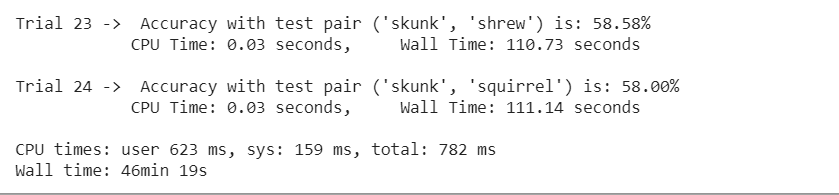


Although magic operator ***%%time*** gave the final time taken, our calculated value is lesser as it excludes loops, print statements and append commands.



* Similarly, testing was performed.





**5.2.2.2.1 Hyper-parameter tuning for the best pair**

* As an additional experiment, we performed the hyperparameter tuning to check if the best pair from logistic regression can improve its lower results with random forest.
* Although results looked promising for Logistic Regression, Random forest had a lower accuracy. We are choosing the best model and tweaking its hyperparameters to improve its accuracy for random forest. Further, we want to know if the accuracy will match once the task is completed.
* For this, we created a **pipeline**, **parameter grid** and **cross-validated** the results with 3 folds to obtain a 2% increase in accuracy.

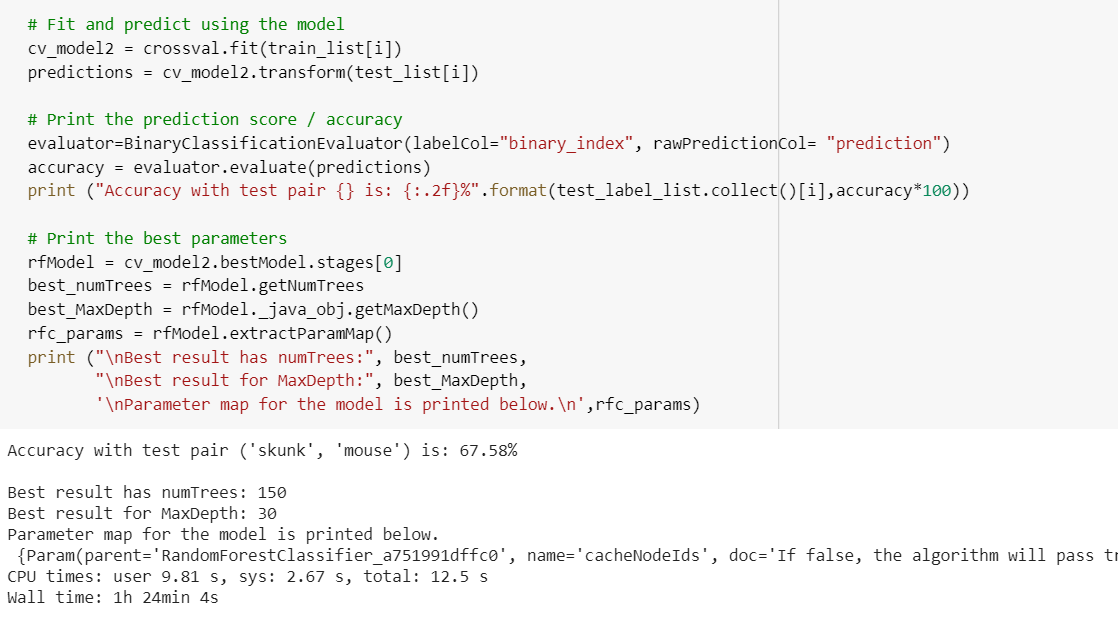
Results of hyperparameter tuning using 3-fold cross validation are as follows.

Random Forest:

* Best 'numTrees': 150, followed by 70
* Best 'maxDepth': 30



We created a one-stage pipeline to experiment and check if it works. A parameter grid was created to perform 3-fold cross validation.



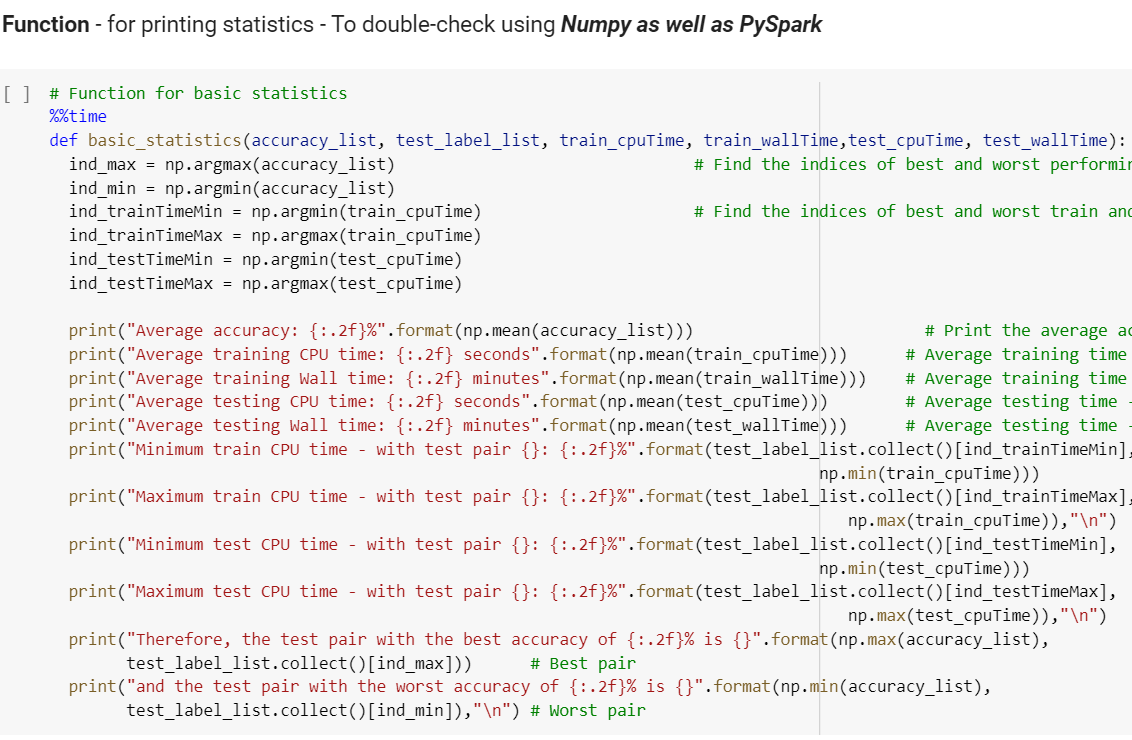
For Random Forest classifier, we tested one pair with 6 values - [10,20,50,70,100,150] for the parameter 'numTrees'. MaxDepth was tested with [1,5,10,20,30]. The model with best accuracy had depth 30 and numTrees 150. We achieved a 2% increase in accuracy. Random forest could not outperform Logistic Regression in terms of accuracy but the results had more consistency due to lower standard deviation.

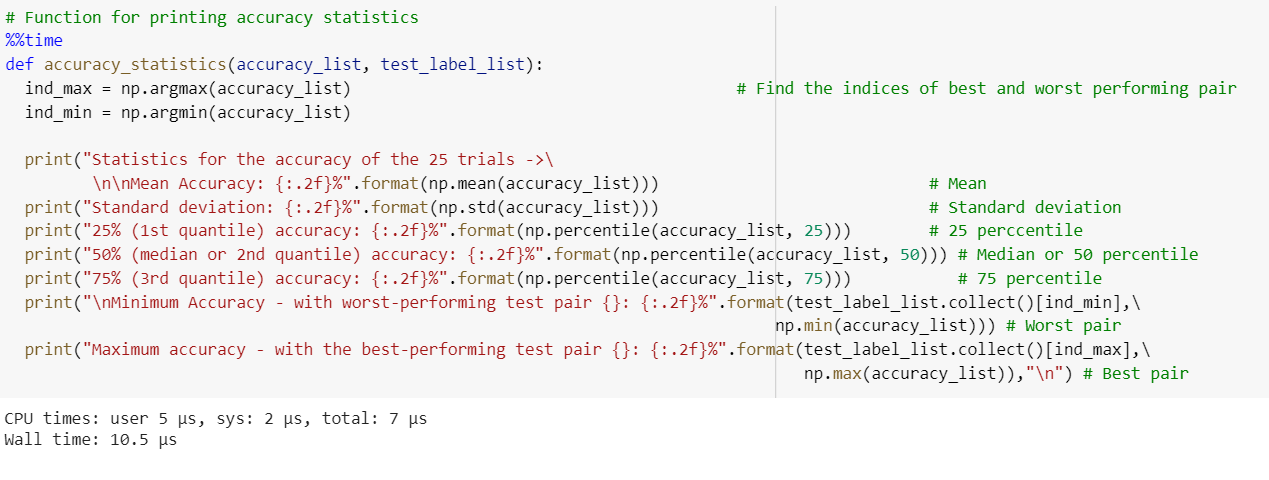
**5.3 Comparison of output statistics with Numpy and Pyspark**

**( Milestone 2 )**

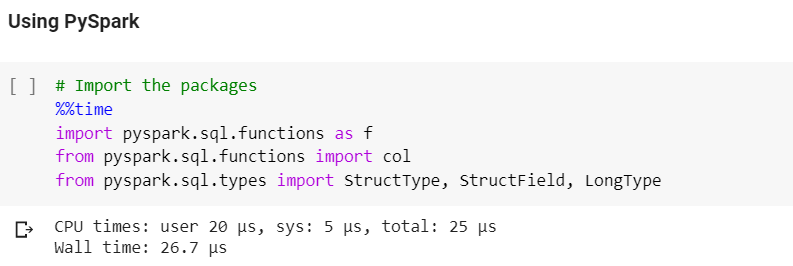
**5.3.1 Logistic Regression**

* Basic statistics, were printed using Numpy and Pyspark to double check the values. Below are the functions created with Numpy.

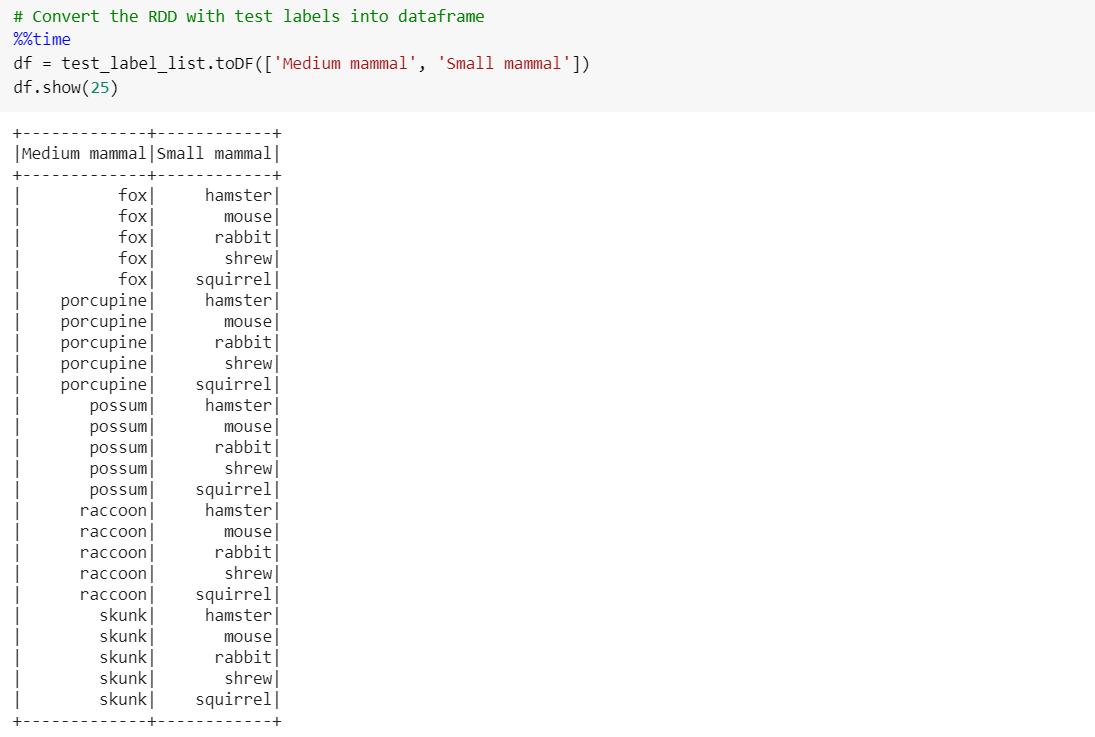




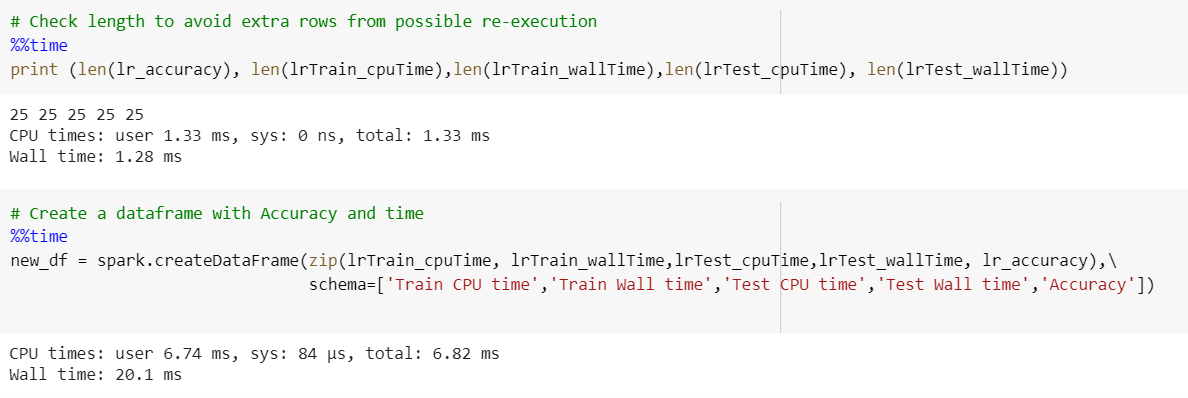
* For Pyspark, we decided to extensively process the results.



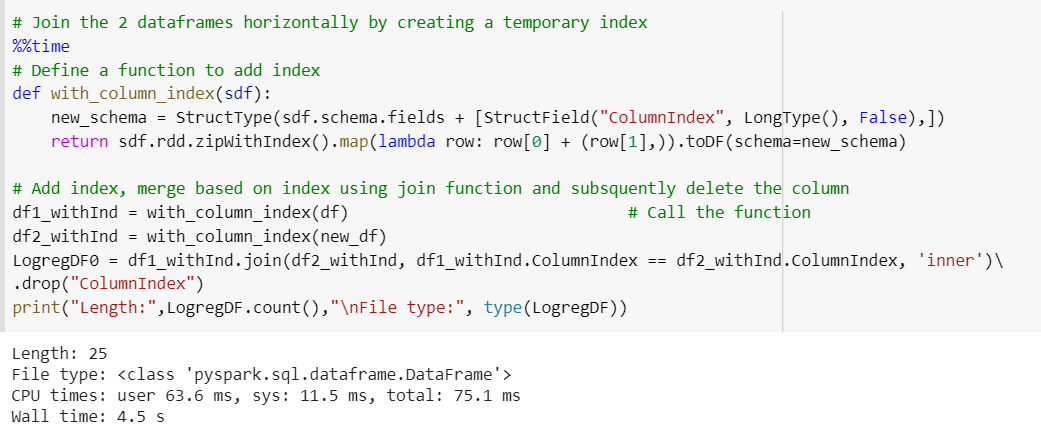
* We converted the RDD with test labels to dataframe.



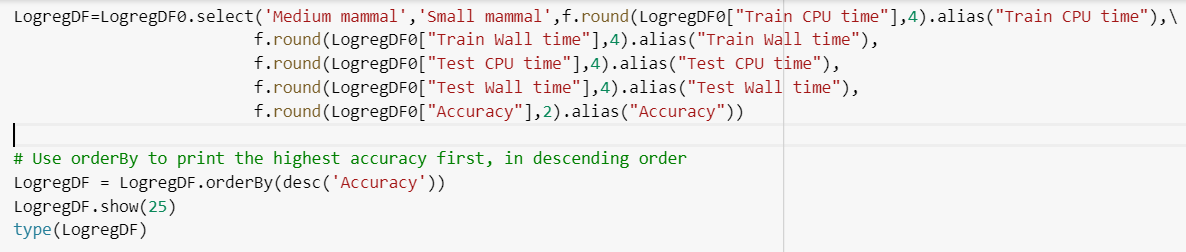
* We created a dataframe with the lists as columns using ***zip()*** function.

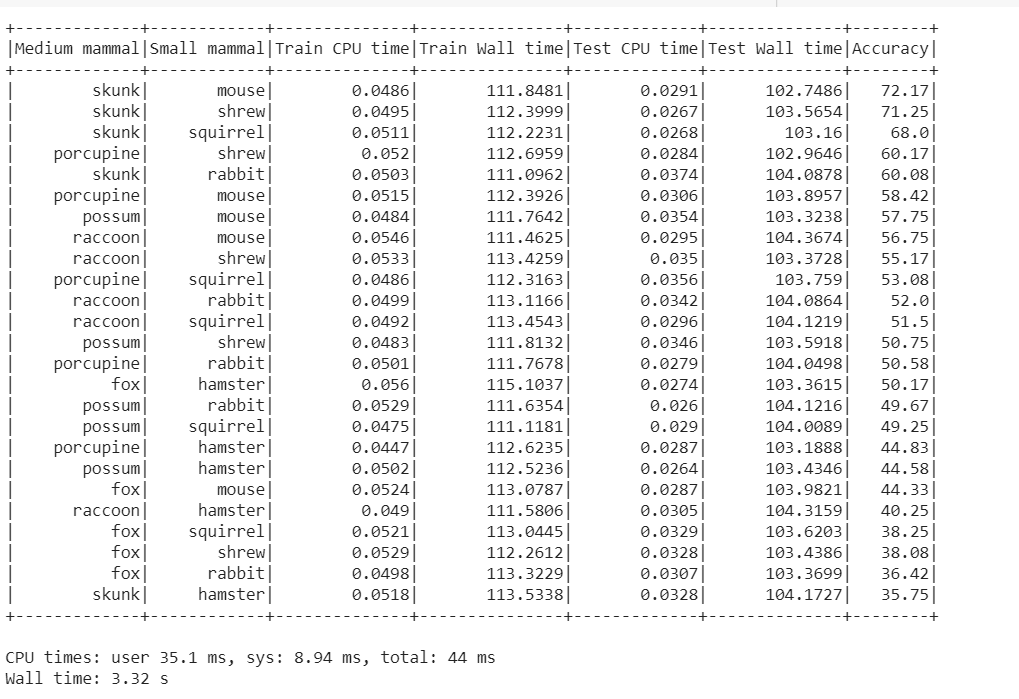


* We joined the tables horizontally using a temporary index number for outer join.

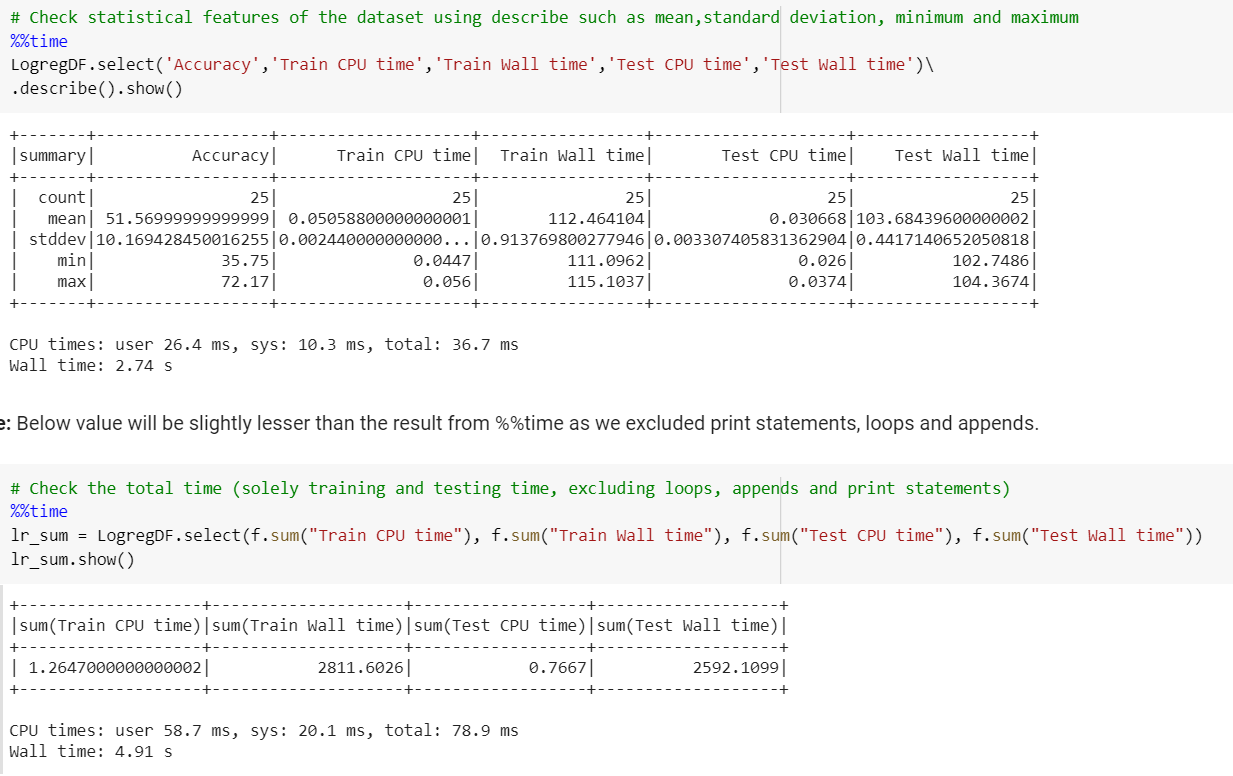


* Thereafter, we selected the columns, round to 2 decimal points using ***round()***function, and displayed the results in descending order using SQL function ***desc.***

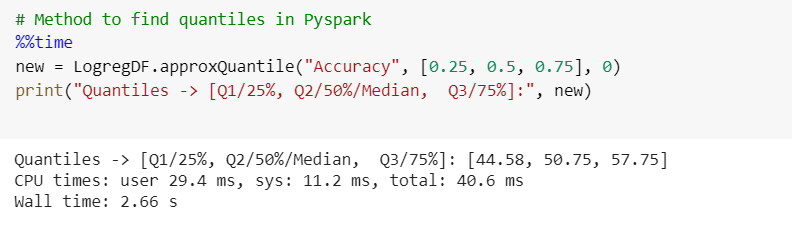




* Further we print statistics using ***describe()*** function and sum using SQL function ***sum().***



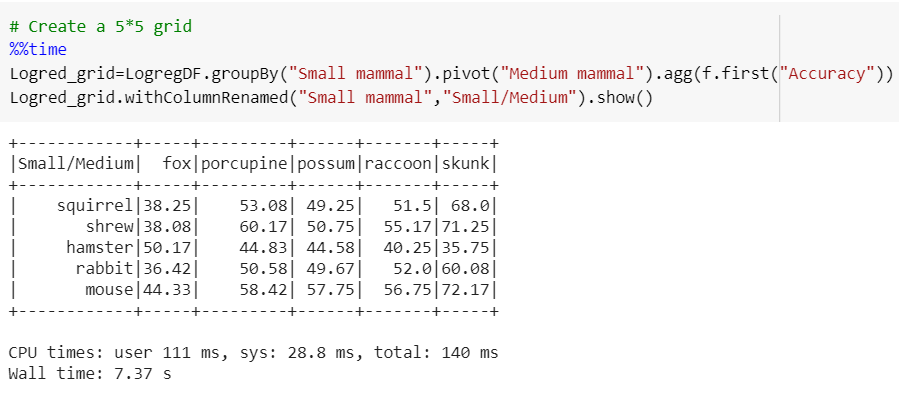
* Quantiles were printed using ***approxquantile()*** function.



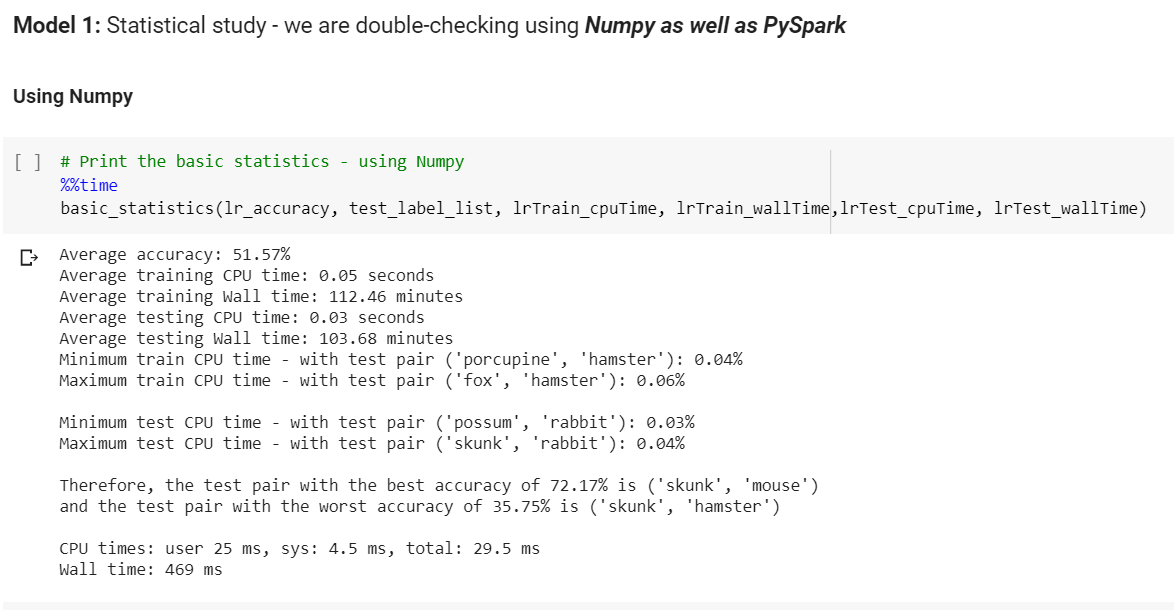
* We merged all results including time and accuracy for each pair, into a 5\*15 grid, using ***groupBy, pivot, aggregate, SQL function first***  and ***alias***(for renaming).



* Here is the final 5\*5 grid for logistic regression.

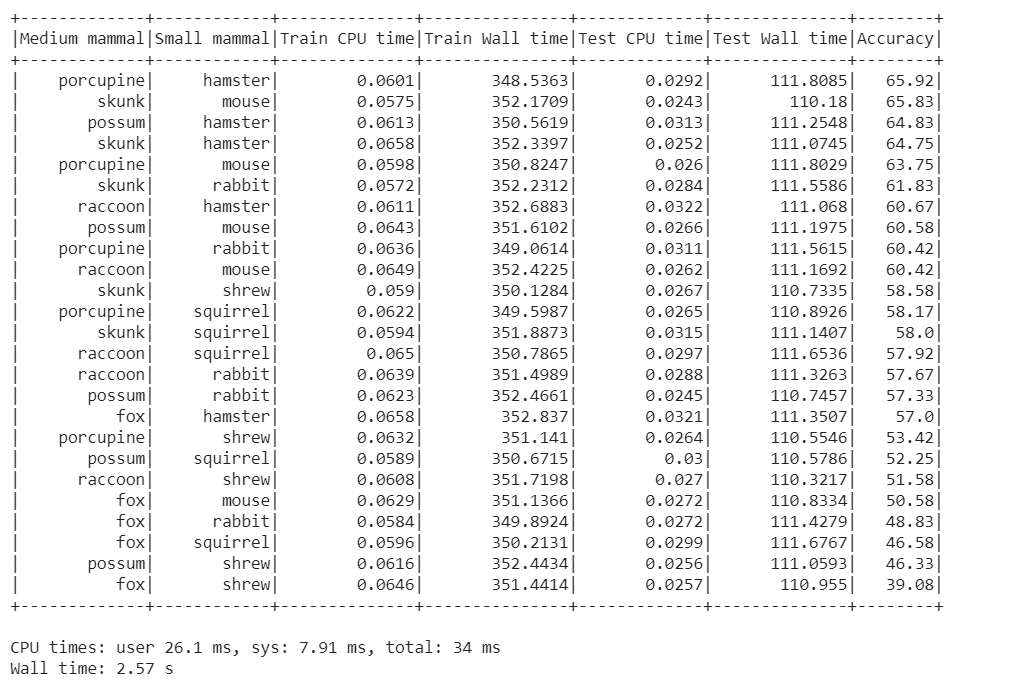


* The results were verified using the function created using Numpy.

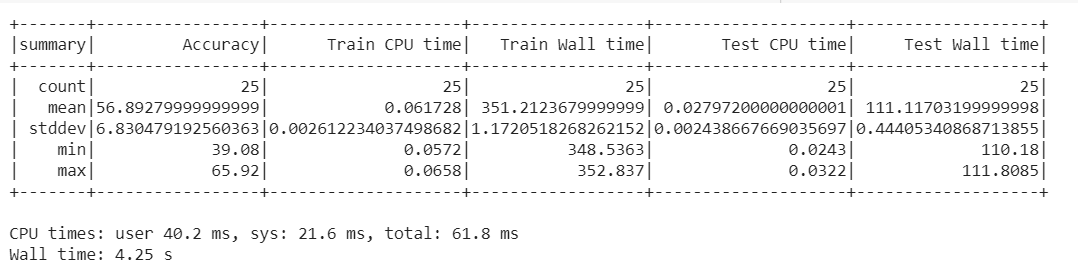


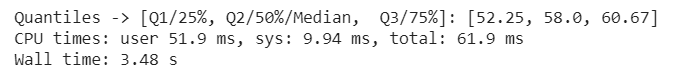
**5.3.2 Random Forest**

* Similar manipulations were performed to derive the below results.

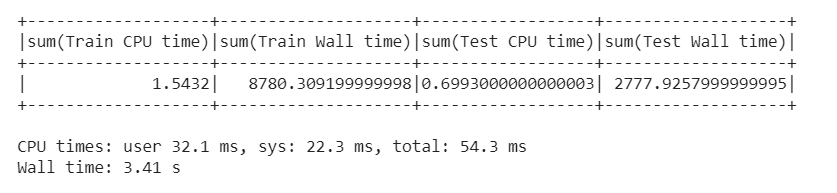


* Statistical features and quantiles are as follows.

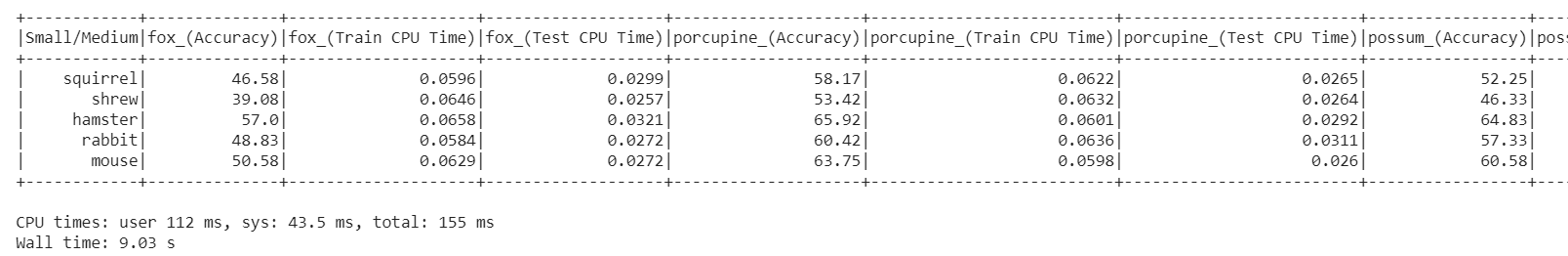




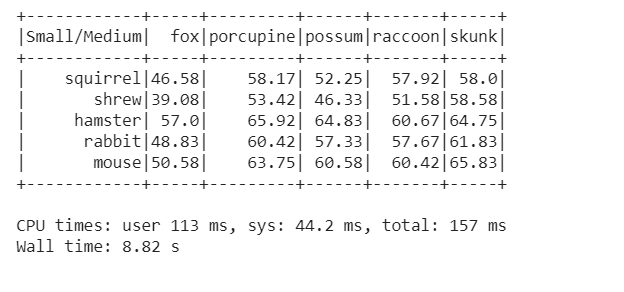
* Sum of CPU and wall times are as follows.



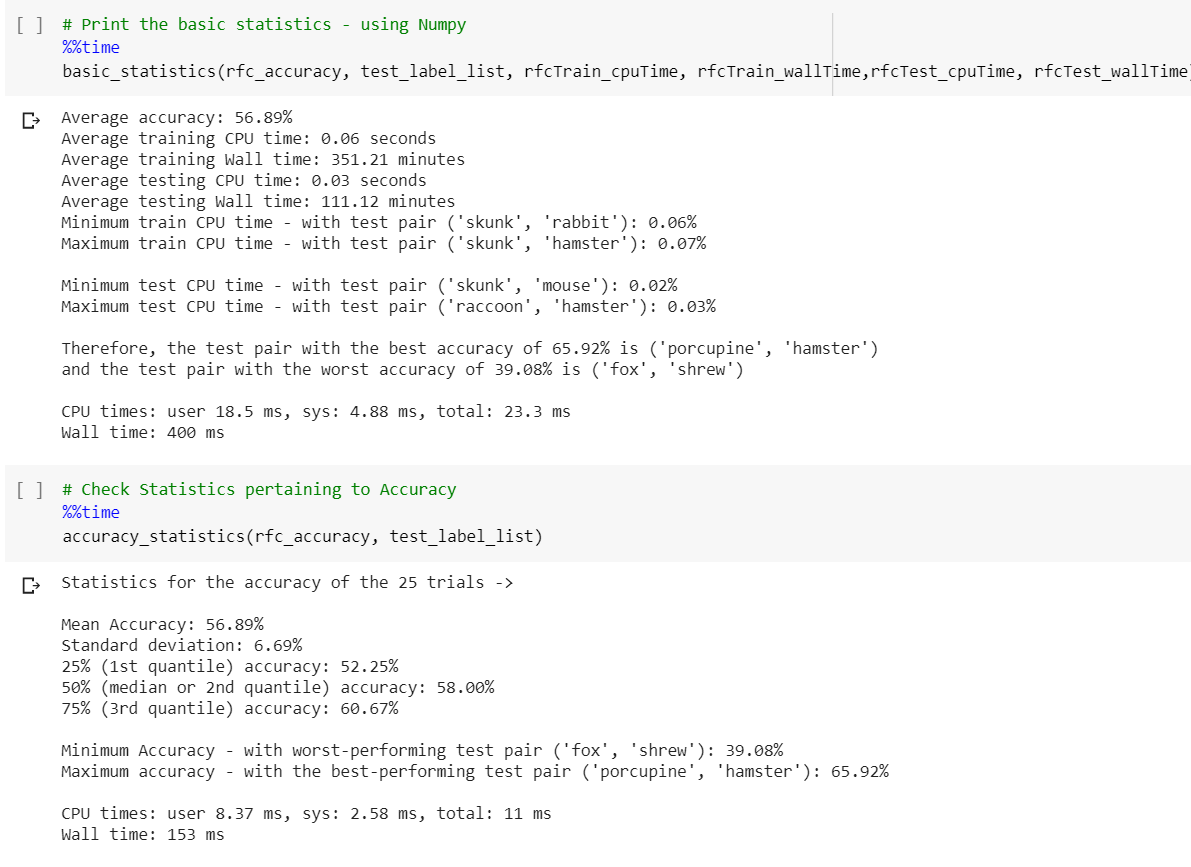
* 5\*15 grid table with accuracy, Train/Test CPU and wall time for each pair.



* 5\*5 grid of pairs with accuracy using Random Forest Classifier.



* Statistics generated by Numpy were exactly the same.



Hence, we learned that PySpark can be used to generate all statistics using populated using Numpy. Although not straightforward or simple as numpy, they accomplish the task with precision.

## 

# Code comparison

We compared codes that have similar functionality for the 2 approaches and listed them in the below table.

**Table of Code Comparison between PySpark MLib and Scikit-learn**

|  |  |  |  |
| --- | --- | --- | --- |
| Steps | | Previous | Current |
| **Data download and preparation** | Resource | from keras.datasets import cifar100 | Download cifar100 python version from <https://www.cs.toronto.edu/~kriz/cifar.html> |
| Read the data | (x\_train, y\_train), (x\_test, y\_test) = cifar100.load\_data() | First, unzip the file and upload the cifar100 to colab;  Second, unpickle the train data and test data;  Third, convert the train and test file to RDD respectively |
| Combine train and test data | np.concatenate | By using union to combine two RDD,  Transform the RDD to DataFrame by using spark.createDataFrame |
| Filter out the assigned superclasses | Generate the target indexes for two superclasses.  By using enumerate function slice the target data index and label | Generate the target indexes for two superclasses(same).  By using filter with isin to filter out the target data  filter(col('fine\_labels').isin(target\_index)) |
| Display the first 5 rows | df.head() | target\_df.show(5) |
| Take a random sample without replacement | df.sample(frac=0.5, replace=True, random\_state=1) | Seed is used to save the state of the random function in subsequent executions.    combine\_rdd.takeSample(withReplacement=False, num=5, seed=123)    We can also use orderBy function with rand to randomly order the dataframe. Further, limit is used to choose the number of rows. df.select([col1, col2]).orderBy(rand()).limit(36).rdd.collect() |
| **Visualize and validate the data** | Validate the data | Normalization and reshape the data  x\_train /= 255.0  x\_test /= 255.0  x\_train.reshape(x\_train.shape[0],3\*32\*32) | Normalization the data/255.0,change the data type to DenseVector()and reshape the data using reshape(3,32,32)(similar) |
| By using np.array and enumerate  function to get the data and the label to do the validation | By using sampleBy function to get the part of the data in a ratio, then pick 4 samples for each class  sampleBy('fine', fractions) |
| Visualize the data | Matplotlib  plt.imshow(img)    Define grid function add the fine label and coarse label | matplotlib(same)  plt.imshow(img)    Add two columns coarse and fine label to the dataFrame, generate the picture with the label easily. |
| **Modify the format for model building step** | Convert coarse label ‘small mammals’ and ‘medium mammals’ into a binary label ‘0.0’ And ‘1.0’. | np.array([[int(y[0] in medium\_sized\_mammals\_index)] for y in y\_train ] | stringindexer = StringIndexer(inputCol='coarse\_labels', outputCol='binary\_index')    target\_name\_df = stringindexer.fit(target\_name\_df).transform(target\_name\_df) |
| Rename column ‘data’ as ‘features’ and re-order columns | df.rename(columns={"Data": "features"})  df = df[['features', 'binary\_index', 'coarse\_labels', 'coarse', 'fine\_labels',’fine]] | target\_name\_df = target\_name\_df.withColumnRenamed("data","features").select("features","binary\_index", "coarse\_labels","coarse","fine\_labels","fine") |
| Check for data type of each column | df.dtypes | target\_name\_df.dtypes    Similar utility was used in spark and pandas. |
| Check for null values in each column | Use isna(), isnull() or isnull().sum(). | for c in target\_name\_df.columns:    print ("Column",c, "- no.of null values:", target\_name\_df.where(col(c).isNull()).count())  The absence of a straightforward null value check was highly inconvenient. We wanted to check in greater granularity and decided to find the null values in each column. It took more than 9 minutes to execute. |
| **Generate random Train-test split** | Randomly select 80% of data as training data and remaining 20% data as testing data. Seed is used to save the state of the random function in subsequent executions. | from sklearn.model\_selection import train\_test\_split  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42) | train\_df, test\_df= target\_name\_df.randomSplit([0.80,0.20], seed=1369) |
| Verify the number of entries in training and testing data | X\_train.shape  x\_test.shape | train\_df.count()  test\_df.count()    We can use spark sql functions too. |
| Verify the distinct labels | np.unique() | train\_df.select('coarse\_labels','binary\_index','coarse').distinct().collect()    train\_df.select('coarse\_labels','binary\_index','coarse').distinct().collect() |
| **Split train and test**  **(one missing pair)** | Pick the rows of missing (testing) pair | List implementation to generate the rows of testing pair. Then slicing. | filter(sub\_class\_index == test\_pair\_index) |
| Generate the rows of training data | List implementation to generate the rows of training pair. Then slicing. | filter(sub\_class\_index != test\_pair\_index) |
| Generate list of missing pair label name | append() | sc.parallelize(missing pair label\_name)  union() |
| **Verify the input data** | Check the type of file | type(df) | type(train\_df)    Similar utility was used in spark and pandas. |
| Check the schema | df.info() | train\_df.printSchema() |
| Check the statistical features of the dataset | df.describe() | train\_df.describe().show() |
| **Define a function for model prediction visualization** | Function to print out randomly chosen images and their labels from predictions | Matplotlib  plt.imshow(img)    Define grid function add the fine label and coarse label | samples = predictions.select(['coarse', 'prediction','fine','binary\_index','features']).orderBy(rand()).limit(36).rdd.collect()    matplotlib  plt.imshow(img)    Sample of 36 images, number restricted using limit function, is randomly generated using rand() function. The user-defined function for image with the label is defined using matplotlib (same as in previous milestone). Incorrect predictions were labeled in red using a parameter color as shown below.    if label == pred:          plt.title("Predicted: {}\n Actual: {}".format(pred,label))        else:          plt.title("Predicted: {}\n Actual: {}".format(pred,label), color = "red") |
| **Model 1 – Logistic Regression** | Import package | From sklearn.linear\_model import LogisticRegression | from pyspark.ml.classification import LogisticRegression |
| Instantiate the model | lr = LogisticRegression() | lr = LogisticRegression(labelCol="binary\_index",featuresCol="features",maxIter=10) |
| Fit the model | lr.fit(x\_train\_1, y\_train\_bin) | model=lr.fit(train\_df) |
| Predict using the model and print the first 10 rows from the resultant dataframe. | lr\_pred=lr.predict(x\_test\_1)    cifar\_grid(x\_test\_1, y\_test\_bin,indices,4,lr\_pred) | predict\_lr=model.transform(test\_df)    predict\_lr.select("coarse","binary\_index","prediction","probability").show(10)    Total execution time for this model was more than 2 minutes. |
| Print the prediction score / accuracy | print ("Logistic Regression Accuracy: {}%".format(lr.score(x\_test\_1, y\_test\_bin)\*100)) | BinaryClassificationEvaluator’  or multiclassclassificationevaluator can be used for finding the accuracy of the 3 models.    eval=BinaryClassificationEvaluator(labelCol="binary\_index", rawPredictionCol= "prediction")    accuracy = (eval.evaluate(predict\_lr))\*100  print("Model Accuracy: %.3f%%" % accuracy) |
| **Model 2 – Naïve Bayes Classifier** | Import package | from sklearn.naive\_bayes import MultinomialNB | from pyspark.ml.classification import NaiveBayes |
| Instantiate the model | naive = MultinomialNB() | naive\_bayes = NaiveBayes(featuresCol="features", labelCol="binary\_index",smoothing=1.0, modelType="multinomial") |
| Fit the model | naive.fit(x\_train\_1, y\_train\_bin) | naive\_bayes = naive\_bayes.fit(train\_df) |
| Predict using the model and print the first 10 rows from the resultant dataframe. | naive\_predict= naive.predict(x\_test\_1)    cifar\_grid(x\_test\_1, y\_test\_bin,indices,4,naive\_predict) | predict\_nb = naive\_bayes.transform(test\_df)    predict\_nb.select("coarse","binary\_index","prediction","probability").show(10)    Total execution time for this model was more than 3 minutes. |
| **Model 3 – Random Forest Classifier** | Import package | from sklearn.ensemble import RandomForestClassifier | from pyspark.ml.classification import RandomForestClassifier |
| Instantiate the model | logit = RandomForestClassifier() | rfc=RandomForestClassifier(featuresCol="features", labelCol="binary\_index",numTrees=100) |
| Fit the model | logit.fit(x\_train\_1, y\_train\_bin) | rfc\_model=rfc.fit(train\_df) |
| Predict using the model and print the first 10 rows from the resultant dataframe. | logit\_pred=logit.predict(x\_test\_1)    cifar\_grid(x\_test\_1, y\_test\_bin,indices,4,logit\_pred) | predict\_rfc=rfc\_model.transform(test\_df)  predict\_rfc.select("coarse","binary\_index","prediction","probability").show(10)    Total execution time for this model was more than 8 minutes. |
| **Confusion matrix and classification report** | Confusion matrix | confusion\_matrix(y\_test\_bin, logit\_pred, labels=None, sample\_weight=None) | **Method 1** -Simple version using below code by converting the dataframe to RDD and using zipWithIndex and countByKey functions.    conf\_mat1 = predict\_lr.select("binary\_index","prediction")  print (conf\_mat1.rdd.zipWithIndex().countByKey())    **Method 2** - Formatted version using below code using multiclassmetrics.    predictionRDD\_1 = predict\_lr.select(['binary\_index', 'prediction']) \  .rdd.map(lambda line: (line[1], line[0]))    metrics1 = MulticlassMetrics(predictionRDD\_1)    cm1 = metrics1.confusionMatrix().toArray()    Thereafter, it is printed using for loop. The absence of a ready-made function made this step inconvenient as it took nearly 6 minutes for both methods to execute in pyspark. |
| Classification report | classification\_report(y\_test\_bin, logit\_pred) | Below function was created using ‘Multiclassmetrics’ to print the classification  report with precision, recall,   f1-score and support    def cr1(label\_in):    precision = metrics1.precision(label=label\_in)    recall = metrics1.recall(label=label\_in)    F1\_Measure = metrics1.fMeasure(label=label\_in)    support = test\_df.filter(test\_df.binary\_index==label\_in).count()    print("%10s %12.2f  %12.2f %12.2f %12d" % \          (label\_in,precision, recall, F1\_Measure, support))    Thereafter, a for loop was created to format and print the report. Similar, to confusion matrix, lack of function made this step inconvenient. It took nearly 6-7 minutes to execute. |
| Processing for generating Statistics | Merging lists into dataframe | pd.DataFrame( {'List': lst1, 'List 2': lst2}) | We use zip() function in spark.  spark.createDataFrame(zip(rfcTrain\_cpuTime, rfcTrain\_wallTime,rfcTest\_cpuTime,rfcTest\_wallTime, rfc\_accuracy),\  schema=['Train CPU time','Train Wall time','Test CPU time','Test Wall time','Accuracy']) |
| Merging two dataframes | pd.concat([df1, df4], axis=1, sort=False) | We create a temporary index.  from pyspark.sql.types import StructType, StructField, LongType  # Define a function to add index  def with\_column\_index(sdf):  new\_schema = StructType(sdf.schema.fields + [StructField("ColumnIndex", LongType(), False),])  return sdf.rdd.zipWithIndex().map(lambda row: row[0] + (row[1],)).toDF(schema=new\_schema)  # Add index, merge based on index using join function and subsquently delete the column  df1\_withInd1 = with\_column\_index(df) # Call the function  df2\_withInd1 = with\_column\_index(new\_df1)  RForest\_DF1 = df1\_withInd1.join(df2\_withInd1, df1\_withInd1.ColumnIndex == df2\_withInd1.ColumnIndex, 'inner').drop("ColumnIndex") |
| Rounding decimal points | Round () function | SQL function needs to be imported to use round() function. |
| For Pivoting the table to create 5\*5 or 5\*15 grid. | pd.concat([df1, df4], axis=1, join='inner') | pivot() function with groupBy() function can be used in PySpark |
| For calculating quantiles | .quantile() function is used in Pandas. | approxQuantile() function is used in PySpark.  RForest\_DF.approxQuantile("Accuracy", [0.25, 0.5, 0.75], 0) |

*Table 3.4 Table of Code Comparison between PySpark MLib and Scikit-learn*

# Results - with comparison

### 

## 7.1 Logistic Regression

## 

### 7.1.1 Milestone 1 - Prediction on Randomly Selected Testing Images Results

The comparison of Confusion Matrix and Classification Report for Scikit-Learn vs. Spark for our Milestone 1 results are shown below:

### Logistic Regression Confusion Matrix Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scikit-Learn (Last semester)** | | | **PySpark** | | |
|  | predict 0 | predict 1 |  | predict 0 | predict 1 |
| real 0 | 293 | 207 | real 0 | 357 | 143 |
| real 1 | 197 | 303 | real 1 | 229 | 271 |

Table 5.a.1 Table of Logistic Regression Confusion Matrix Comparison

### 

* Classification report for both Scikit-Learn and PySpark is shown below

|  |  |
| --- | --- |
| **Scikit-Learn** | **PySpark** |
|  |  |

* We can see that PySpark model performs slightly better at eliminating False Negatives than Scikit-Learn. But both models have around 60% accuracy.

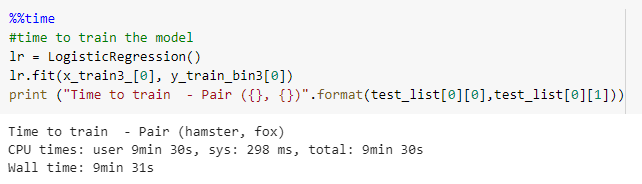
### 7.1.2 Milestone 2 - Prediction on one testing subclass images from each of the two superclasses Results

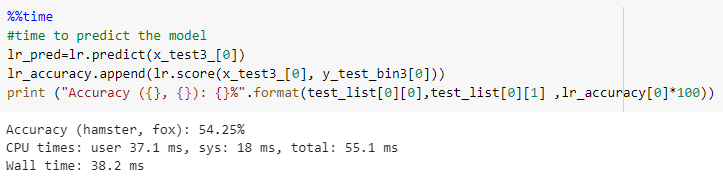
#### 7.1.2.1 Scikit-Learn Results

In order to adequately compare the performance of Scikit-Learn and PySpark, part of the work done last semester was updated. Using the dataset prepared with the missing pair, the logistic regression model was rerun, now timing two separate steps: Training and Prediction.

Through this exercise it was detected, as expected, that most of the time goes to training the model with the prediction happening in mili-seconds.

The code to train and predict is very straightforward, as the sample below shows.

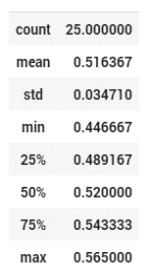




This routine was rerun for every possible combination of missing paring and the table below summarizes the findings.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Logistic regression**  **(last semester)** | **Scikit-Learn** | Hamster | Mouse | Rabbit | Shrew | Squirrel |
| Fox | Score | 54.25% | 48.50% | 46.25% | 44.60% | 44.83% |
| Training  CPU time | 9m30s | 7m57s | 5m45s | 2m55s | 3m8s |
| Prediction  CPU time (ms) | 55.1 | 32.7 | 33.8 | 33.6 | 34.1 |
| Porcupine | Score | 52.00% | 55.00% | 50.58% | 54.36% | 54.00% |
| Training  CPU time | 10m57s | 3m55s | 5m19s | 8m22s | 7m28s |
| Prediction  CPU time (ms) | 30.4 | 39.7 | 41.2 | 38.5 | 39.2 |
| Possum | Score | 48.50% | 54.50% | 50.50% | 48.92% | 50.58% |
| Training  CPU time | 4m8s | 11m21s | 7m5s | 3m48s | 10m30s |
| Prediction  CPU time (ms) | 42.3 | 44 | 27.2 | 37.3 | 46 |
| Raccoon | Score | 51.08% | 54.83% | 52.83% | 53.58% | 51.42% |
| Training  CPU time | 15m9s | 10m | 4m43s | 3m9s | 3m38s |
| Prediction  CPU time (ms) | 35.3 | 34.2 | 41.6 | 37.5 | 45 |
| Skunk | Score | 47.67% | 56.49% | 56.16% | 55.33% | 54.08% |
| Training  CPU time | 16m6s | 5m29s | 3m2s | 11m5s | 6m31s |
| Prediction  CPU time (ms) | 50.6 | 33.5 | 35.1 | 36.8 | 47.1 |

Just as before, Fox remains the worse category, being the one hardest to predict when not a part of training. In this run of the code the standard deviation was much smaller than previous runs, but the mean value remained in the low 50s. The figure below shows the statistics.



Regarding the training time, on average it took 434s, with a standard deviation of 222s. The longest took 966s and the fastest 175s.

#### 7.1.2.2 PySpark Results

Results for 25 pairs using PySpark Logistic Regression is shown below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Logistic regression** | Fox | | | Porcupine | | | Possum | | | Raccoon | | | Skunk | | | AVERAGE |
| **PySpark** | Score | Training  CPU time (s) | Prediction  CPU time (s) | Score | Training  CPU time (s) | Prediction  CPU time (s) | Score | Training  CPU time (s) | Prediction  CPU time (ss) | Score | Training  CPU time (s) | Prediction  CPU time (s) | Score | Training  CPU time | Prediction  CPU time (s) |  |
| Hamster | 50.17% | 0.06 | 0.03 | 44.83% | 0.04 | 0.03 | 44.58% | 0.05 | 0.03 | 40.25% | 0.05 | 0.03 | 35.75% | 0.05 | 0.03 | 43.12% |
| Mouse | 44.33% | 0.05 | 0.03 | 58.42% | 0.05 | 0.03 | 57.75% | 0.05 | 0.04 | 56.75% | 0.05 | 0.03 | 72.17% | 0.05 | 0.03 | 57.88% |
| Rabbit | 36.42% | 0.05 | 0.03 | 50.58% | 0.05 | 0.03 | 49.67% | 0.05 | 0.03 | 52.00% | 0.05 | 0.03 | 60.08% | 0.05 | 0.04 | 49.75% |
| Shrew | 38.08% | 0.05 | 0.03 | 60.17% | 0.05 | 0.03 | 50.75% | 0.05 | 0.03 | 55.17% | 0.05 | 0.03 | 71.25% | 0.05 | 0.03 | 55.08% |
| Squirrel | 38.25% | 0.05 | 0.03 | 53.08% | 0.05 | 0.04 | 49.25% | 0.05 | 0.03 | 51.50% | 0.05 | 0.03 | 68.00% | 0.05 | 0.03 | 52.02% |
| AVERAGE | 41.45% | 0.052 | 0.03 | 53.42% | 0.048 | 0.032 | 50.40% | 0.05 | 0.032 | 51.13% | 0.05 | 0.03 | 61.45% | 0.05 | 0.032 |  |

Statistical results are shown below:



Observation:

* We can see that for Logistic Regression, PySpark has significantly larger standard deviation than Scikit-Learn. PySpark has lower minimum value and higher maximum value than Scikit-Learn, indicating Scikit-Learn Logistic Regression is more robust than the PySpark one.
* High/low accuracy distribution for the 2 methods exhibit the same pattern, which means the pairs that have the lowest accuracies in the Scikit-Learn table are also the lowest in the PySpark table. This shows that both methods have similar qualitative prediction power.

## 7.2 Random Forest

## 

### 7.2.1 Milestone 1 - Prediction on Randomly Selected Testing Images Results

The comparison of Confusion Matrix and Classification Report for Scikit-Learn vs. Spark for our Milestone 1 results are shown below:

### Random ForestClassifier Confusion Matrix Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scikit-Learn (Last semester)** | | | **PySpark** | | |
|  | predict 0 | predict 1 |  | predict 0 | predict 1 |
| real 0 | 361 | 139 | real 0 | 383 | 173 |
| real 1 | 244 | 256 | real 1 | 209 | 378 |

Table 5.c.1 Table of Random Forest Classifier Confusion Matrix Comparison

● Classification report for both Scikit-Learn and PySpark is shown below:

|  |  |
| --- | --- |
| **Scikit-Learn** | **PySpark** |
|  |  |

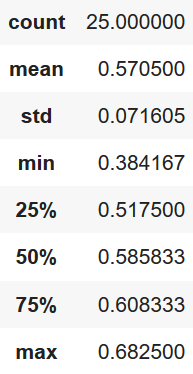
● We can see that PySpark have more balance in predicting the binary labels, but Scikit-Learn predicts 1 label better than 0 labels. During training, accuracy score for Random Forest can vary from 62% to 70%, showing the model is not very robust.

### 7.2.2 Milestone 2 - Prediction On One Testing Subclass Images From Each of the Two Superclasses Results

#### 7.2.2.1 Scikit-Learn Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Random Forest** | **Scikit-Learn** | Hamster | Mouse | Rabbit | Shrew | Squirrel | Average accuracy |
| Fox | Score | 56.67% | 53.00% | 49.58% | 38.42% | 46.25% | 48.78% |
| Training  CPU time (ms) | 1.47s | 1.48s | 1.46s | 1.49s | 1.46s |  |
| Prediction  CPU time (ms) | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s |  |
| Porcupine | Score | 65.00% | 62.67% | 59.83% | 51.42% | 60.33% | 59.85% |
| Training  CPU time (ms) | 1.44s | 1.45s | 1.49s | 1.46s | 1.46s |  |
| Prediction  CPU time (ms) | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s |  |
| Possum | Score | 63.33% | 60.83% | 59.83% | 46.75% | 49.83% | 56.11% |
| Training  CPU time (ms) | 1.45s | 1.48s | 1.46s | 1.47s | 1.47s |  |
| Prediction  CPU time (ms) | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s |  |
| Raccoon | Score | 59.17% | 60.25% | 57.92% | 51.75% | 57.83% | 57.38% |
| Training  CPU time (ms) | 1.44s | 1.46s | 1.46s | 1.49s | 1.47s |  |
| Prediction  CPU time (ms) | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s |  |
| Skunk | Score | 67.50% | 68.25% | 63.83% | 60.25% | 57.00% | 63.37% |
| Training  CPU time | 1.46s | 1.46s | 1.46s | 1.48s | 1.45s |  |
| Prediction  CPU time (ms) | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s |  |
|  | Average Accuracy | 61.04% | 59.19% | 56.79% | 47.09% | 53.56% | 55.53% |

Statistics for the accuracy scores were:



#### 7.2.2.2 PySpark Results

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Random Forest** | Fox | | | Porcupine | | | Possum | | | Raccoon | | | Skunk | | | AVERAGE |
| **PySpark** | Score | Training  CPU time (ms) | Prediction  CPU time (ms) | Score | Training  CPU time (ms) | Prediction  CPU time (ms) | Score | Training  CPU time (ms) | Prediction  CPU time (ms) | Score | Training  CPU time (ms) | Prediction  CPU time (ms) | Score | Training  CPU time | Prediction  CPU time (ms) |  |
| Hamster | 57.00% | 0.07 | 0.03 | 65.92% | 0.06 | 0.03 | 64.83% | 0.06 | 0.03 | 60.67% | 0.06 | 0.03 | 64.75% | 0.07 | 0.03 | 62.63% |
| Mouse | 50.58% | 0.06 | 0.03 | 63.75% | 0.06 | 0.03 | 60.58% | 0.06 | 0.03 | 60.42% | 0.06 | 0.03 | 65.83% | 0.06 | 0.02 | 60.23% |
| Rabbit | 48.83% | 0.06 | 0.03 | 60.42% | 0.06 | 0.03 | 57.33% | 0.06 | 0.03 | 57.67% | 0.06 | 0.03 | 61.83% | 0.06 | 0.03 | 57.22% |
| Shrew | 39.08% | 0.06 | 0.03 | 53.42% | 0.06 | 0.03 | 46.33% | 0.06 | 0.03 | 51.58% | 0.06 | 0.03 | 58.58% | 0.06 | 0.03 | 49.80% |
| Squirrel | 46.58% | 0.06 | 0.03 | 58.17% | 0.06 | 0.03 | 52.25% | 0.06 | 0.03 | 57.92% | 0.06 | 0.03 | 58.00% | 0.06 | 0.03 | 54.58% |
| AVERAGE | 48.41% | 0.062 | 0.03 | 60.34% | 0.06 | 0.03 | 56.26% | 0.06 | 0.03 | 57.65% | 0.06 | 0.03 | 61.80% | 0.062 | 0.028 |  |

Statistical results are shown below:



Observation:

* We can see that for Random Forest, PySpark has significantly larger standard deviation than Scikit-Learn. PySpark has lower minimum value and higher maximum value than Scikit-Learn, indicating Scikit-Learn Random Forest is more robust than the PySpark one.
* High/low accuracy distribution for the 2 methods exhibit the same pattern, which means the pairs that have the lowest accuracies in the Scikit-Learn table are also the lowest in the PySpark table. This shows that both methods have similar qualitative prediction power.

## 7.3 Naive Bayes

## 

### 7.3.1 Milestone 1 - Prediction on Randomly Selected Testing Images Results

### Naïve Bayes Classifier Confusion Matrix Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scikit-Learn (Last semester)** | | | **PySpark** | | |
|  | predict 0 | predict 1 |  | predict 0 | predict 1 |
| real 0 | 311 | 189 | real 0 | 344 | 212 |
| real 1 | 201 | 299 | real 1 | 244 | 343 |

Table 5.b.1 Table of Naïve Bayes Classifier Confusion Matrix Comparison

● Classification report for both Scikit-Learn and PySpark is shown below:

|  |  |
| --- | --- |
| **Scikit-Learn** | **PySpark** |
|  |  |

● We can see that Scikit-Learn Naïve Bayes have more balance in predicting the binary labels, but PySpark predicts 1 label better than 0 labels.

## 7.4 Quantitative comparison

To compare the performance between Spark Machine Learning models and Scikit-Learn, we ran the same models from both libraries on the same dataset for a binary classification problem. All models were run in Google Colab with TPU. By comparing the results, we see that PySpark has faster CPU time than Scikit-learn but longer wall time. This is because we must wait for server response from Spark but can almost instantly run on Python.

For model accuracy, we compared the same models that gave us the best results for Milestone 1 from the previous project. The results did not show significantly better accuracy for one package over the other. We also tested other models such as Extra Decision Tree and SVM, but they have poor accuracy for PySpark.

**Model results overview :**

**Table of Models Results Comparison for Milestone1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Scikit-Learn (Last semester)** | | | **PySpark** | | |
|  | Accuracy | Train time  (CPU time in seconds) | Test time  (CPU time in seconds) | Accuracy | Train time  (CPU time in seconds) | Test time  (CPU time in seconds) |
| **Logistic Regression** | 59.60% | 28.1 s | 49.3 ms | 62.17% | 42.3 ms | 28.9 ms |
| **Naive Bayes** | 61.0% | 169 ms | 61.6 ms | 60.15% | 28.7 ms | 21.4 ms |
| **Random Forest** | 61.2% | 1.93 s | 28.1 ms | 66.64% | 65.7 ms | 21.4 ms |

*Table 7.3.1 Table of Models performance comparison of speed*

**Table of Models Performance Comparison of speed**

|  |  |  |
| --- | --- | --- |
| **Spark/Scikit** | **Logistic Regression (LR) CPU time (sec)** | **Random Forest (RF) CPU time (sec)** |
| **Average training time of 25 trials** | 0.05s / 434.4s | 0.06s /1.46s |
| **Best Spark CPU Time case** | 0.04s / 657s | 0.06s/1.48s |
| **Worst Spark CPU Time case** | 0.06s / 570s | 0.07s/1.46s |

*Table 7.3.2 Table of Models performance comparison of speed*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Spark/Scikit | LR accuracy | LR CPU time(sec) | RF accuracy | RF CPU time(sec) |
| Average prediction time of 25 trials | 51.57%/51.63% | 0.05s/434.4s | 56.89%/55.53% | 0.06/1.46s |
| Best Spark CPU Time case | 44.83%/52.00% | 0.04s/657s | 50.58%/53.00% | 0.06s/1.48s |
| Worst Spark CPU Time case | 50.17%/54.25% | 0.06s/570s | 57.00%/56.67% | 0.07s/1.46s |

*Table 7.3.3 Table of Models performance comparison of accuracy and speed*

# Learning & Insights

# 

The wall time for executing PySpark code in Google Colaboratory turned out to be horrendously long. We explored and researched online before arriving at the below methods, to overcome this bottleneck.

1. Optimize memory usage in system. Empty recycle bin.
2. Optimize browser usage. Remove unwanted plugins, pop-ups, ads, tabs and clear history.
3. Using TensorFlow processing unit (TPU) as runtime type.
4. Increase memory in google colaboratory using below code:

**#spark = SparkSession.builder.master("local[\*]").getOrCreate()**

**Memory\_limit = "12g"**

**spark = SparkSession.builder.appName("Foo").config("spark.executor.memory", Memory\_limit).config("spark.driver.memory", Memory\_limit).getOrCreate()**

Further, we realized that the system keeps reconnecting whenever the colaboratory executes for hours, especially during hyper-parameter tuning. ClickConnect function in the inspect mode of Colaboratory helped in keeping the colab from reconnecting after 12 hours. However, it does not warranty protection from network disruptions.

We can also use alternate methods to cope with runtime disruption such as splitting the execution into 2 sections if it disrupts after more than 10 iterations.

**Other alternatives** - Tried connecting GCP to colab but few packages won’t load. Running colab at night was faster.

**Useful tips we discovered when working with PySpark on Machine Learning tasks:**

* **Computationally intensive** - 25 pair execution of Random forest with 2 hyper-parameters caused colaboratory crash 5 times.
* Increase your memory limit by changing configuration in initial to speed things up.
* If your dataset “fits” your memory, use scikit-learn
* Use Spark once your model is already trained and optimized to run a large set of data
* Pyspark operation is very complex and requires additional steps
* Simple operations such as count(), describe() or union() equivalents are very time consuming
* Input data parameters vary depending on the model
* Combining datasets is very time consuming
* Use Spark sql for count is faster than PySpark in-build method

# 9. Conclusion

In this milestone, we used multiple algorithms to train models to recognize small sized mammals and medium sized mammals with PySpark machine learning models and compared it with results from last semester which were predicted with Scikit-Learn. During the project, we experimented and learned the difference between PySpark MLib, PySpark.ml an Scikit-Learn.

Several of the major differences we learned are:

1. PySpark machine learning models runs faster in CPU time than Scikit-Learn. But in practice, Spark takes longer wall time since our large dataset had longer queue time from server.
2. 25-pair results from PySpark Logistic Regression and Random Forest has a greater standard deviation than that of Scikit-Learn. This may be an indication that the Scikit-Learn models are more robust than that of PySpark. But on the other hand, PySpark can produce single highest result, this may be of use in some cases.
3. We also experimented with pipelining for both PySpark and Scikit-Learn. For Scikit-Learn, the pipelining procedure is relatively simple but did not provide vast improvement on computation speed. For PySpark, the wall time is very long and pipelining sometimes get stuck. We can only do 2-3 parameters.
4. For hyperparameter tuning, both PySpark and Scikit-Learn took a long time to complete. For Random Forest, both methods took around 1 and a half hours. But PySpark sometimes got stuck and had to take up to 4 or 5 hours.
5. Scikit-Learn fits better with our working habit. Scikit-Learn is a well-developed machine learning package with consistent parameter formats and rich ecosystem while PySpark machine learning models requires extra steps to perform some of the basic operations in Scikit-Learn. Input data parameters are also inconsistent for MLib. Some hyperparameters are not common between PySpark and Scikit-Learn.
6. Some simple operations are time consuming in Spark, e.g. combining datasets, ***count(), describe()*** and ***union()*** equivalents. A faster workaround for count is to use Spark SQL.
7. Spark is ideal for running simple models on large datasets with Scala as its programming language on a RAM-rich hardware. Our Laptop computer experienced some memory issues while running Spark. When your RAM can handle the amount of data, then Spark would be a good choice to perform machine learning;
8. A good workflow we concluded for machine learning on large scale data is: first train and optimize your model on a small data sample with Scikit learn, then run the large dataset using Spark.

In general, doing Machine Learning with PySpark has been an educative experience. Through the practice, we learned the difference in design philosophy can reflect on the products of different systems. For instance, Spark is designed as an Analytics Engine for Big Data, so it performs better on large scale data. Scikit-Learn is designed as a python machine-learning library, so it provides a better machine learning workflow.

# 

# Appendix:

## References

Opala, M. (2018). *Top Machine Learning Frameworks Compared: Scikit-Learn, Dlib, MLib, Tensorflow, and More*. Retrieved from netguru.com: https://www.netguru.com/blog/top-machine-learning-frameworks-compared

Quesada, J., & Anderson, D. (2016). *Data Science Retreat*. Retrieved from https://www.slideshare.net/JoseQuesada5/a-full-machine-learning-pipeline-in-scikitlearn-vs-in-scalaspark-pros-and-con

Ruusmann, V. (2017). *Openscouring.IO*. Retrieved from <https://www.slideshare.net/VilluRuusmann/r-scikitlearn-and-apache-spark-ml-what-difference-does-it-make>

## Referred Resources

<https://spark.apache.org/docs/2.3.0/ml-classification-regression.html>

<https://spark.apache.org/docs/2.3.0/ml-tuning.html>

<https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-mllib/spark-mllib-BinaryClassificationEvaluator.html>

<https://stackoverflow.com/questions/41714698/how-to-get-accuracy-precision-recall-and-roc-from-cross-validation-in-spark-ml>

<https://drive.google.com/file/u/0/d/1ZJwp9Oyp5adE7J57XEj_OOdYLDPO_q3B/edit>

<https://drive.google.com/file/u/0/d/1kennf0tB893aLbIRY8CD4tmo-nbErLQx/edit>

<https://stackoverflow.com/questions/48202900/what-does-these-parameters-mean-in-jupyter-notebook-when-i-input-time>

<http://spark.apache.org/docs/latest/api/python/pyspark.ml.html>

<http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html>

<https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/>

<https://www.programcreek.com/python/example/93276/pyspark.ml.classification.LogisticRegression>

<http://benalexkeen.com/multiclass-text-classification-with-pyspark/>

<https://jarrettmeyer.com/2017/05/04/random-forests-with-pyspark>

<https://medium.com/@dhiraj.p.rai/logistic-regression-in-spark-ml-8a95b5f5434c>

<https://docs.databricks.com/applications/machine-learning/mllib/binary-classification-mllib-pipelines.html>

<https://runawayhorse001.github.io/LearningApacheSpark/classification.html>

<https://towardsdatascience.com/multi-class-text-classification-with-pyspark-7d78d022ed35>

<https://towardsdatascience.com/machine-learning-with-pyspark-and-mllib-solving-a-binary-classification-problem-96396065d2aa>