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

TABLE OF CONTENTS

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,	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 these 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

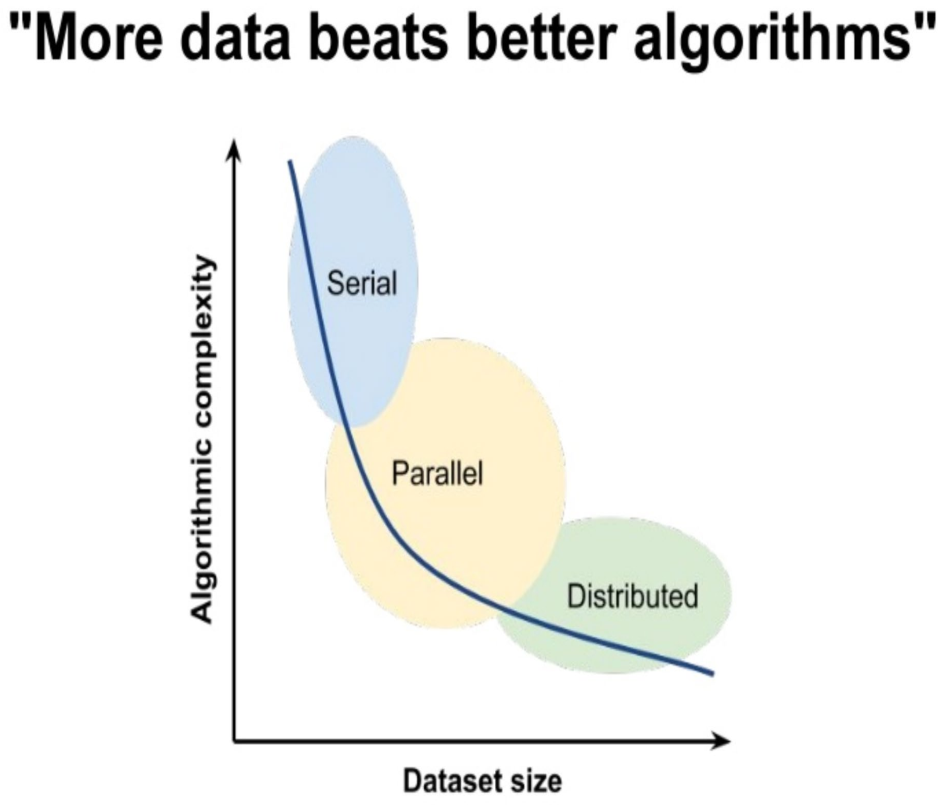


Fig 3.1: Dataset size and algorithmic performance

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

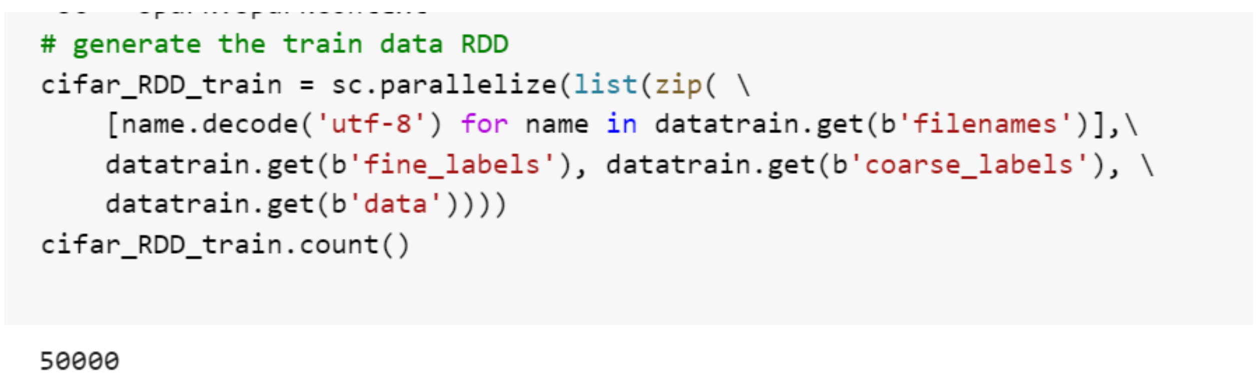
ML with Sckit-learn	**ML with Spark (Python)**
Limited to one machine	Takes a long time to aggregate dataframe (ML Lib not efficient)
Not advisable for extremely large datasets	Memory expensive
	High latency
	Handy simple functions missing for tasks such as classification matrix or printing null values.

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

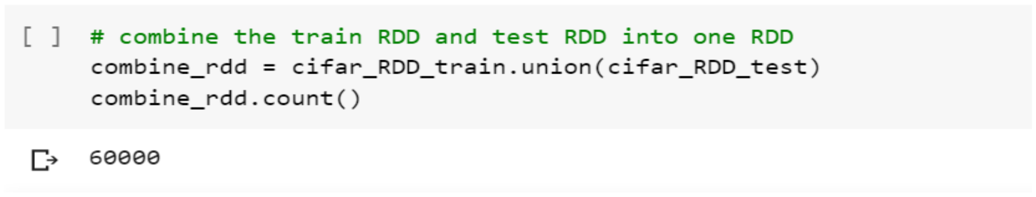
- In order to download data, we installed OpenJDK and findspark packages to set up the environment and build Spark session. Then, we imported all the necessary packages and modules such as Spark SQL and MLib.
- Reading in the data with PySpark is not as straightforward as importing from Keras datasets 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.



```
# generate the train data RDD
cifar_RDD_train = sc.parallelize(list(zip( \
    [name.decode('utf-8') for name in datatrain.get(b'filenames')],\
    datatrain.get(b'fine_labels'), datatrain.get(b'coarse_labels'), \
    datatrain.get(b'data'))))
cifar_RDD_train.count()
```

```
50000
```

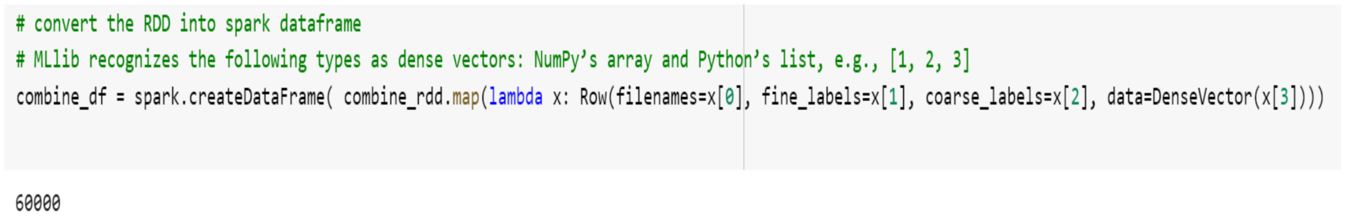
- With ***union*** function, we combined test RDD and train RDD. The total count of records is 60000.



```
[ ]  # combine the train RDD and test RDD into one RDD
     combine_rdd = cifar_RDD_train.union(cifar_RDD_test)
     combine_rdd.count()
```

```
60000
```

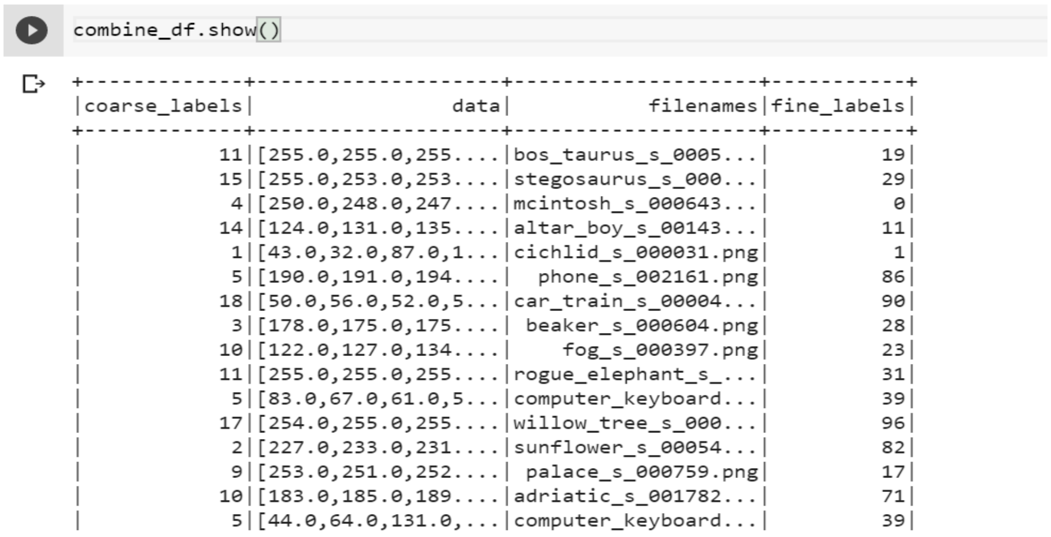
- We transformed the RDD into Spark DataFrame by with ***createDataFrame()*** and ***map()*** function. During this step, we converted the data type to ***DenseVector*** so Spark can recognize them as arrays of value.



```
# convert the RDD into spark dataframe
# MLlib recognizes the following types as dense vectors: NumPy’s array and Python’s list, e.g., [1, 2, 3]
combine_df = spark.createDataFrame( combine_rdd.map(lambda x: Row(filenames=x[0], fine_labels=x[1], coarse_labels=x[2], data=DenseVector(x[3]))))
```

```
60000
```

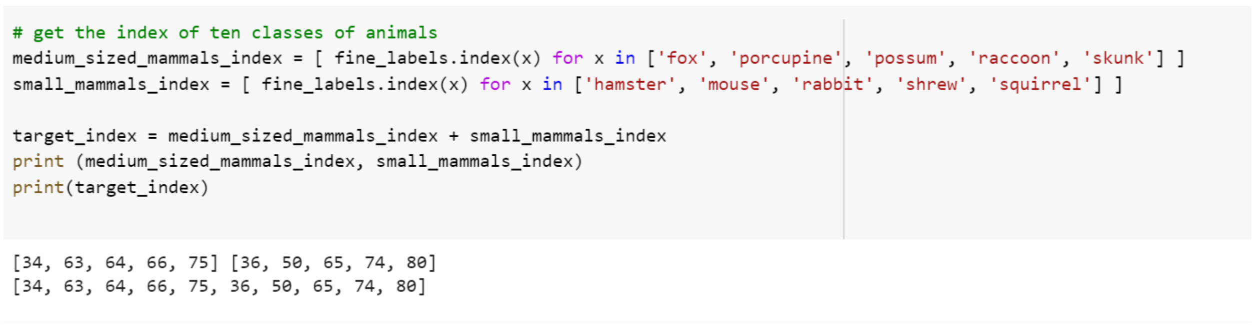
- In our DataFrame, there are four columns, ***filename, coarse label, fine label and data***.



```
combine_df.show()
```

```
+-------------+-------------------+-------------------+-----------+
|coarse_labels|               data|          filenames|fine_labels|
+-------------+-------------------+-------------------+-----------+
|           11|[255.0,255.0,255....|bos_taurus_s_0005...|         19|
|           15|[255.0,253.0,253....|stegosaurus_s_000...|         29|
|            4|[250.0,248.0,247....|mcintosh_s_000643...|          0|
|           14|[124.0,131.0,135....|altar_boy_s_00143...|         11|
|            1|[43.0,32.0,87.0,1...|cichlid_s_000031.png|          1|
|            5|[190.0,191.0,194....|  phone_s_002161.png|         86|
|           18|[50.0,56.0,52.0,5...|car_train_s_00004...|         90|
|            3|[178.0,175.0,175....| beaker_s_000604.png|         28|
|           10|[122.0,127.0,134....|    fog_s_000397.png|         23|
|           11|[255.0,255.0,255....|rogue_elephant_s_...|         31|
|            5|[83.0,67.0,61.0,5...|computer_keyboard...|         39|
|           17|[254.0,255.0,255....|willow_tree_s_000...|         96|
|            2|[227.0,233.0,231....|sunflower_s_00054...|         82|
|            9|[253.0,251.0,252....| palace_s_000759.png|         17|
|           10|[183.0,185.0,189....|adriatic_s_001782...|         71|
|            5|[44.0,64.0,131.0,...|computer_keyboard...|         39|
```

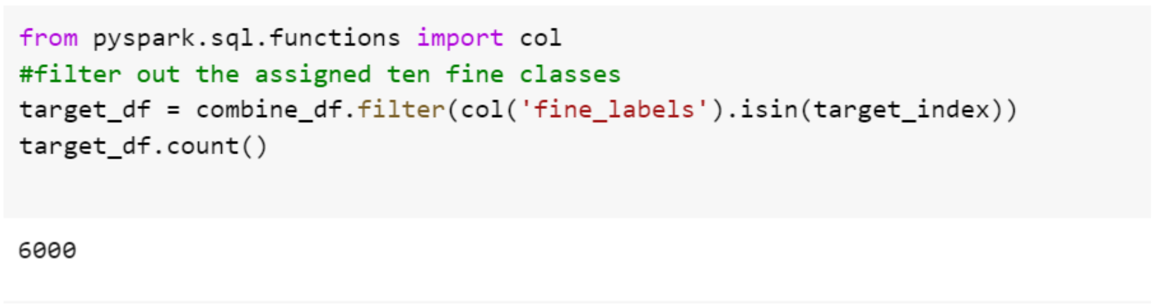
- We generated indexes of 2 superclasses as a list. The assigned superclasses (small mammals and medium-sized mammals) were filtered by using given metadata.



```
# get the index of ten classes of animals
medium_sized_mammals_index = [ fine_labels.index(x) for x in ['fox', 'porcupine', 'possum', 'raccoon', 'skunk'] ]
small_mammals_index = [ fine_labels.index(x) for x in ['hamster', 'mouse', 'rabbit', 'shrew', 'squirrel'] ]

target_index = medium_sized_mammals_index + small_mammals_index
print (medium_sized_mammals_index, small_mammals_index)
print(target_index)
```

```
[34, 63, 64, 66, 75] [36, 50, 65, 74, 80]
[34, 63, 64, 66, 75, 36, 50, 65, 74, 80]
```

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

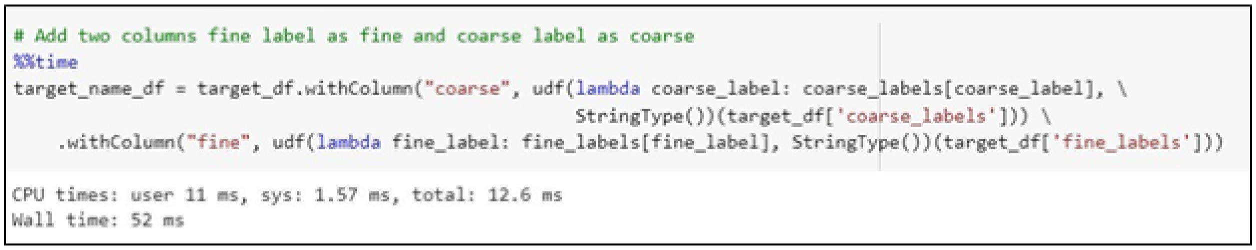


```
from pyspark.sql.functions import col
#filter out the assigned ten fine classes
target_df = combine_df.filter(col('fine_labels').isin(target_index))
target_df.count()
```

```
6000
```

4.2 Data Validation

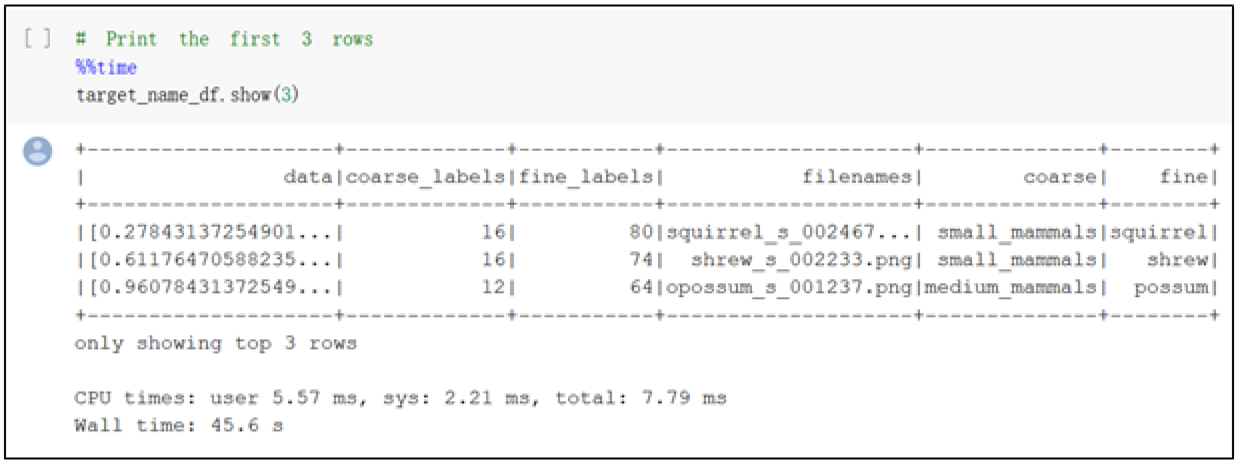
● We validated our data on the generated DataFrame. With ***.withColumn()*** function, we added two columns ***coarse*** and ***fine*** which contain the verbal labels for each record.



```
# Add two columns fine label as fine and coarse label as coarse
%%time
target_name_df = target_df.withColumn("coarse", udf(lambda coarse_label: coarse_labels[coarse_label], \
                                                    StringType())(target_df['coarse_labels'])) \
    .withColumn("fine", udf(lambda fine_label: fine_labels[fine_label], StringType())(target_df['fine_labels']))
```

```
CPU times: user 11 ms, sys: 1.57 ms, total: 12.6 ms
Wall time: 52 ms
```

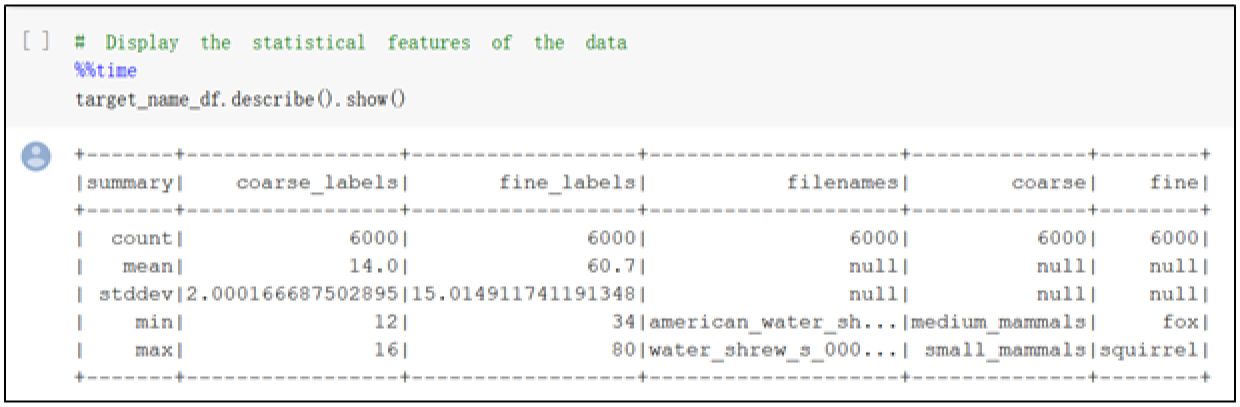
● Let's check if the columns were created correctly.



```
[ ] # Print the first 3 rows
    %%time
    target_name_df.show(3)
```

```
+--------------------+-------------+-----------+--------------------+--------------+--------+
|                data|coarse_labels|fine_labels|           filenames|        coarse|    fine|
+--------------------+-------------+-----------+--------------------+--------------+--------+
|[0.27843137254901...|           16|         80|squirrel_s_002467...| small_mammals|squirrel|
|[0.61176470588235...|           16|         74|   shrew_s_002233.png| small_mammals|   shrew|
|[0.96078431372549...|           12|         64|opossum_s_001237.png|medium_mammals|  possum|
+--------------------+-------------+-----------+--------------------+--------------+--------+
only showing top 3 rows

CPU times: user 5.57 ms, sys: 2.21 ms, total: 7.79 ms
Wall time: 45.6 s
```

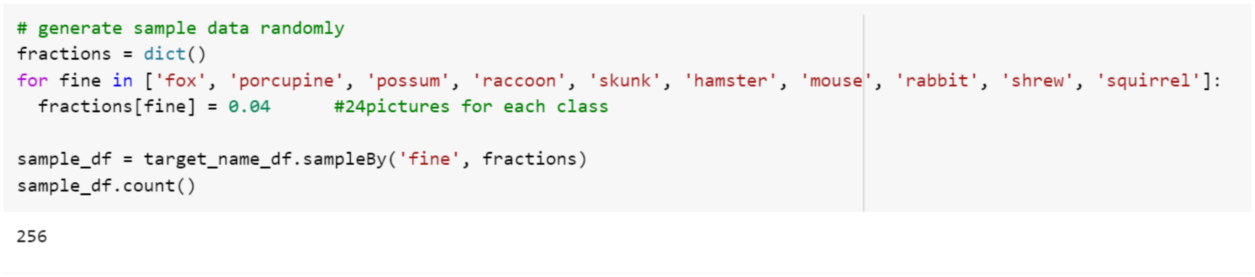
● Check statistical features of the data.



```
[ ]  # Display the statistical features of the data
     %%time
     target_name_df.describe().show()
```

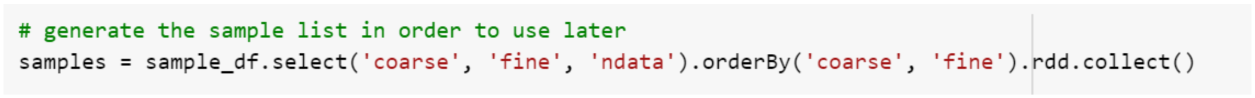
```
+-------+-----------------+-----------------+--------------------+-------------+--------+
|summary|    coarse_labels|      fine_labels|           filenames|       coarse|    fine|
+-------+-----------------+-----------------+--------------------+-------------+--------+
|  count|             6000|             6000|                6000|         6000|    6000|
|   mean|             14.0|             60.7|                null|         null|    null|
| stddev|2.000166687502895|15.014911741191348|               null|         null|    null|
|    min|               12|               34|american_water_sh...|medium_mammals|     fox|
|    max|               16|               80|water_shrew_s_000...| small_mammals|squirrel|
+-------+-----------------+-----------------+--------------------+-------------+--------+
```

- 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.



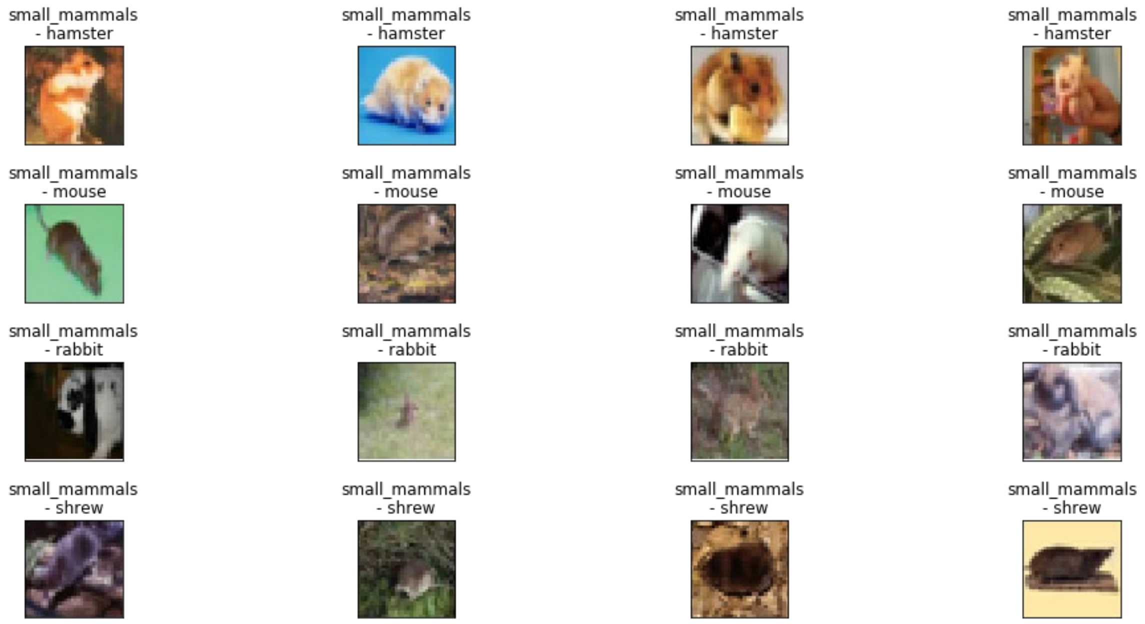
```
# generate sample data randomly
fractions = dict()
for fine in ['fox', 'porcupine', 'possum', 'raccoon', 'skunk', 'hamster', 'mouse', 'rabbit', 'shrew', 'squirrel']:
  fractions[fine] = 0.04      #24pictures for each class

sample_df = target_name_df.sampleBy('fine', fractions)
sample_df.count()
```

```
256
```



```
# generate the sample list in order to use later
samples = sample_df.select('coarse', 'fine', 'ndata').orderBy('coarse', 'fine').rdd.collect()
```

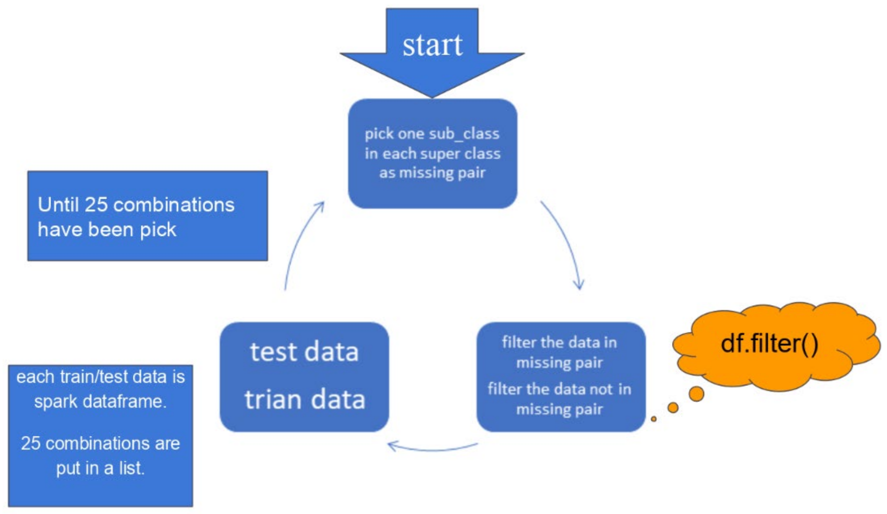
- Visualization was an essential step towards validation of the filtered data. We generated the 4 random images from each subclass using ***matplotlib***. As instructed, the title included the verbal superclass name (coarse label) and subclass name (fine label). The following were the validation images,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

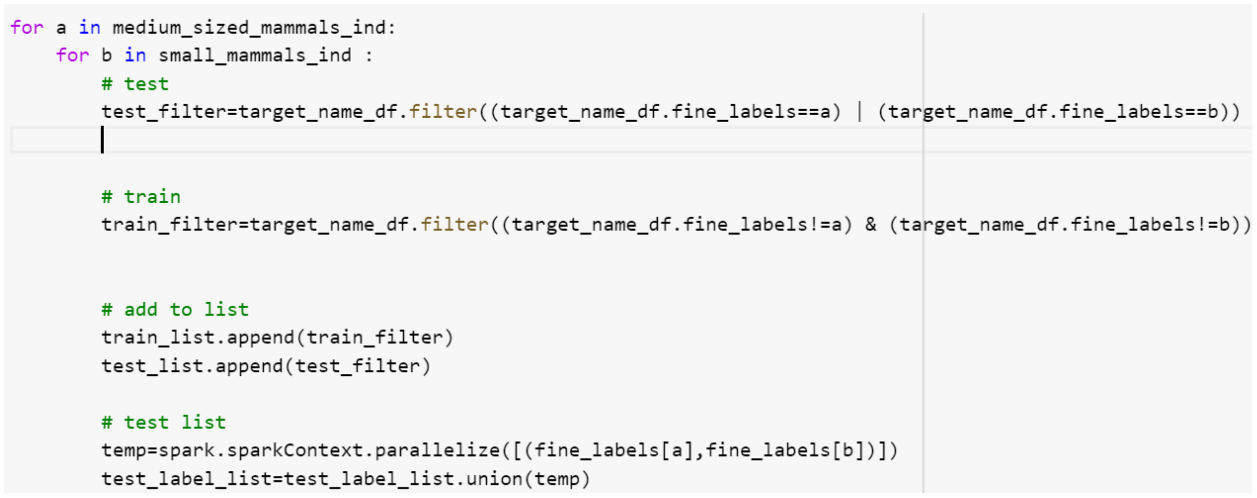


- Once the dataset was validated and ready, we proceeded to model training.

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.


start
pick one sub_class in each super class as missing pair
filter the data in missing pair
filter the data not in missing pair
df.filter()
test data
trian data
Until 25 combinations have been pick
each train/test data is spark dataframe.
25 combinations are put in a list.



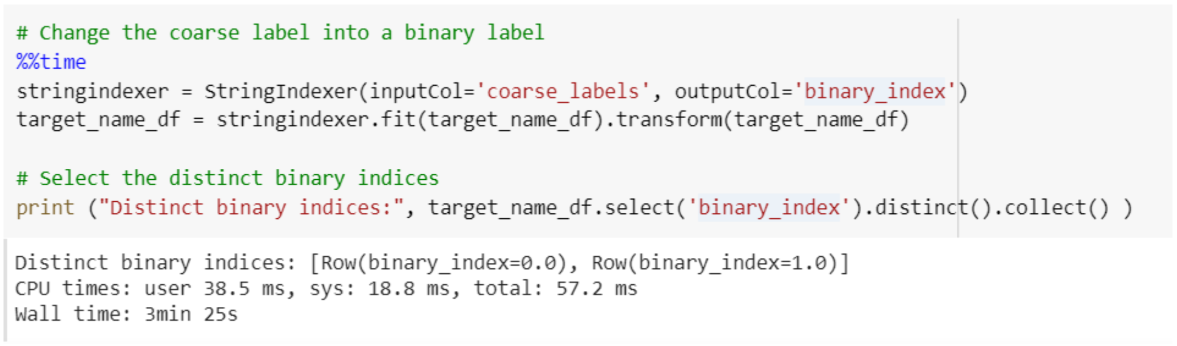
```
for a in medium_sized_mammals_ind:
    for b in small_mammals_ind :
        # test
        test_filter=target_name_df.filter((target_name_df.fine_labels==a) | (target_name_df.fine_labels==b))
        

        # train
        train_filter=target_name_df.filter((target_name_df.fine_labels!=a) & (target_name_df.fine_labels!=b))


        # add to list
        train_list.append(train_filter)
        test_list.append(test_filter)

        # test list
        temp=spark.sparkContext.parallelize([(fine_labels[a],fine_labels[b])])
        test_label_list=test_label_list.union(temp)
```

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.

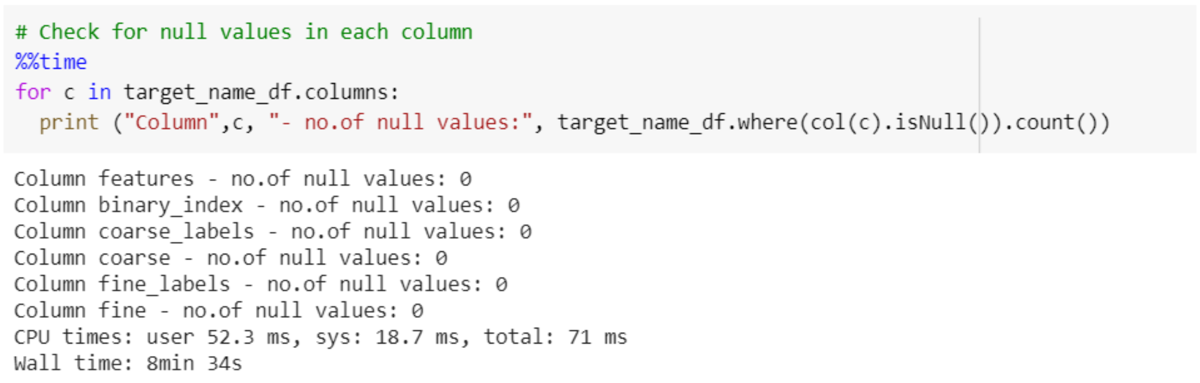
- As we are defining our binary classification problem, we first convert and combine the coarse labels of each superclass into a binary label using ***StringIndexer*** function.



```
# Change the coarse label into a binary label
%%time
stringindexer = StringIndexer(inputCol='coarse_labels', outputCol='binary_index')
target_name_df = stringindexer.fit(target_name_df).transform(target_name_df)

# Select the distinct binary indices
print ("Distinct binary indices:", target_name_df.select('binary_index').distinct().collect() )
```

```
Distinct binary indices: [Row(binary_index=0.0), Row(binary_index=1.0)]
CPU times: user 38.5 ms, sys: 18.8 ms, total: 57.2 ms
Wall time: 3min 25s
```

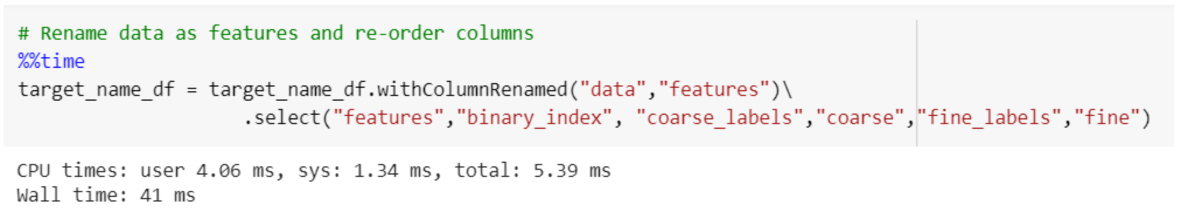
- Data integrity was ensured by checking null values in the dataset, which were absent.



```
# Check for null values in each column
%%time
for c in target_name_df.columns:
  print ("Column",c, "- no.of null values:", target_name_df.where(col(c).isNull()).count())
```

```
Column features - no.of null values: 0
Column binary_index - no.of null values: 0
Column coarse_labels - no.of null values: 0
Column coarse - no.of null values: 0
Column fine_labels - no.of null values: 0
Column fine - no.of null values: 0
CPU times: user 52.3 ms, sys: 18.7 ms, total: 71 ms
Wall time: 8min 34s
```

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

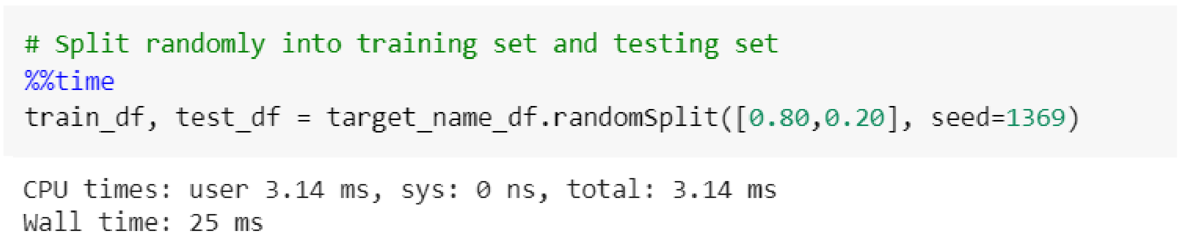


```
# Rename data as features and re-order columns
%%time
target_name_df = target_name_df.withColumnRenamed("data","features")\
                    .select("features","binary_index", "coarse_labels","coarse","fine_labels","fine")
```

```
CPU times: user 4.06 ms, sys: 1.34 ms, total: 5.39 ms
Wall time: 41 ms
```

5.1.1 Milestone 1

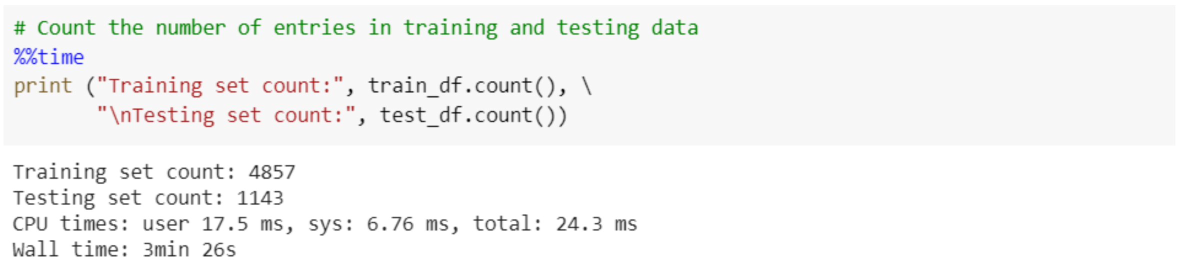
- The dataset was split into training and testing sets using ***randomSplit*** function. Here, we randomly select 80% of data as training data and remaining 20% data as testing data. Seed is used to save the state of the random function in subsequent executions.



```
# Split randomly into training set and testing set
%%time
train_df, test_df = target_name_df.randomSplit([0.80,0.20], seed=1369)
```

```
CPU times: user 3.14 ms, sys: 0 ns, total: 3.14 ms
Wall time: 25 ms
```

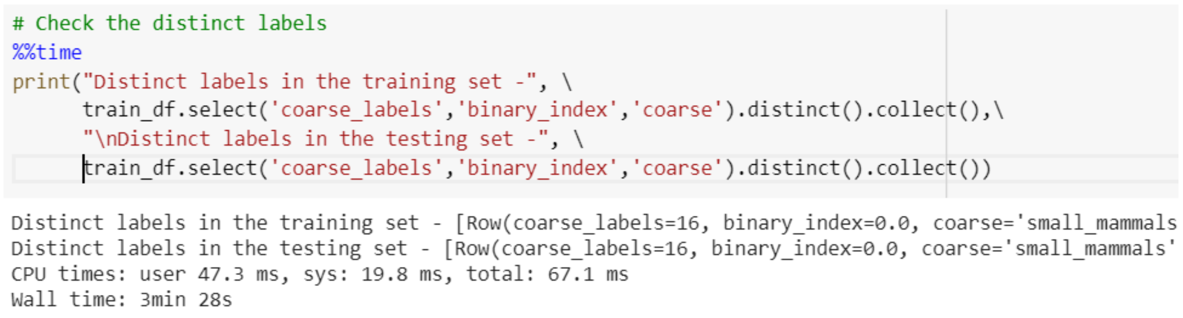
- Training and testing counts were checked using ***.count()*** function.



```
# Count the number of entries in training and testing data
%%time
print ("Training set count:", train_df.count(), \
       "\nTesting set count:", test_df.count())
```

```
Training set count: 4857
Testing set count: 1143
CPU times: user 17.5 ms, sys: 6.76 ms, total: 24.3 ms
Wall time: 3min 26s
```

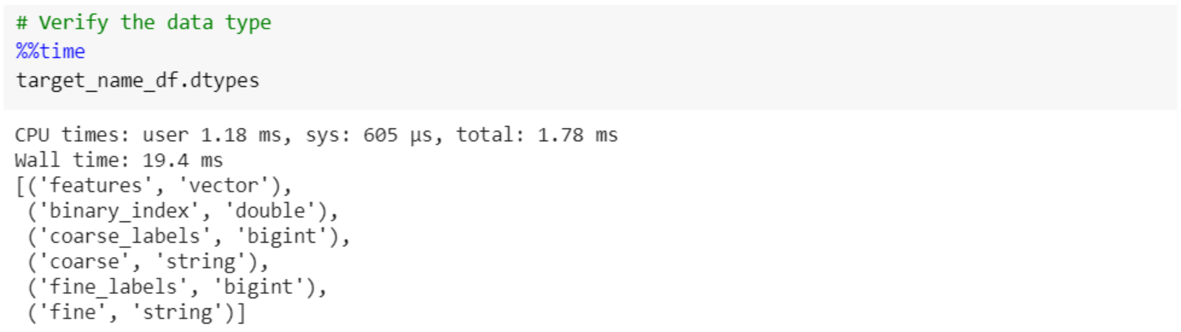
- Distinct superclass labels were checked by using ***.distinct()*** function.



```
# Check the distinct labels
%%time
print("Distinct labels in the training set -", \
      train_df.select('coarse_labels','binary_index','coarse').distinct().collect(),\
      "\nDistinct labels in the testing set -", \
      train_df.select('coarse_labels','binary_index','coarse').distinct().collect())
```

```
Distinct labels in the training set - [Row(coarse_labels=16, binary_index=0.0, coarse='small_mammals
Distinct labels in the testing set - [Row(coarse_labels=16, binary_index=0.0, coarse='small_mammals'
CPU times: user 47.3 ms, sys: 19.8 ms, total: 67.1 ms
Wall time: 3min 28s
```

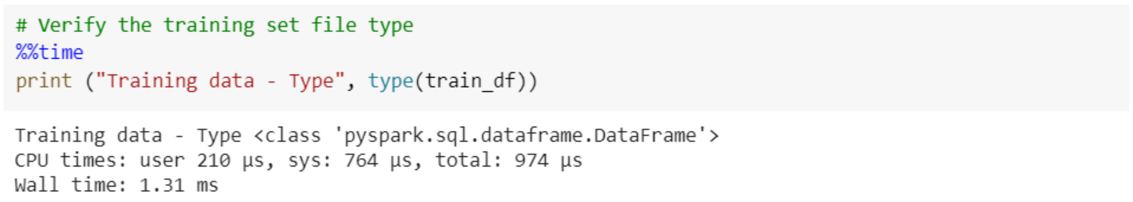
- Datatypes of the final file was checked to ensure adherence with the specified data format, which is vector image data and binary labels.



```
# Verify the data type
%%time
target_name_df.dtypes
```

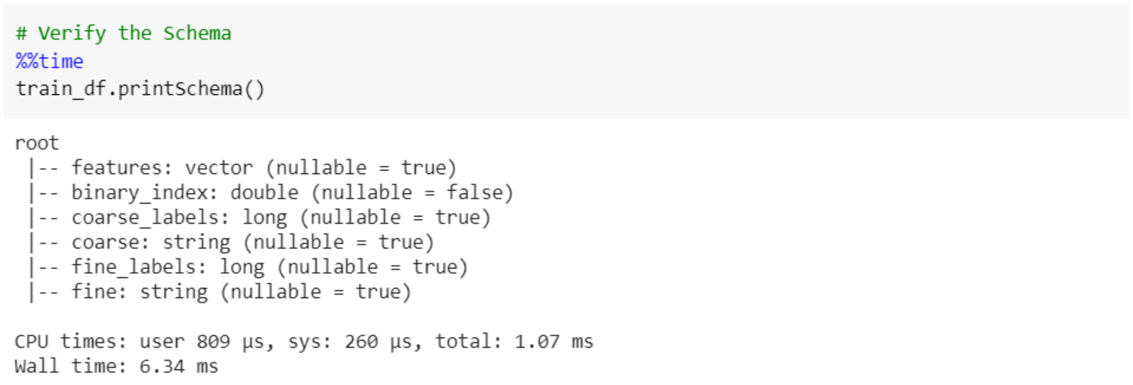
```
CPU times: user 1.18 ms, sys: 605 µs, total: 1.78 ms
Wall time: 19.4 ms
[('features', 'vector'),
 ('binary_index', 'double'),
 ('coarse_labels', 'bigint'),
 ('coarse', 'string'),
 ('fine_labels', 'bigint'),
 ('fine', 'string')]
```

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



```
# Verify the training set file type
%%time
print ("Training data - Type", type(train_df))
```

```
Training data - Type <class 'pyspark.sql.dataframe.DataFrame'>
CPU times: user 210 µs, sys: 764 µs, total: 974 µs
Wall time: 1.31 ms
```



```
# Verify the Schema
%%time
train_df.printSchema()
```

```
root
 |-- features: vector (nullable = true)
 |-- binary_index: double (nullable = false)
 |-- coarse_labels: long (nullable = true)
 |-- coarse: string (nullable = true)
 |-- fine_labels: long (nullable = true)
 |-- fine: string (nullable = true)

CPU times: user 809 µs, sys: 260 µs, total: 1.07 ms
Wall time: 6.34 ms
```

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

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

```
# Display the statistical features
%%time
train_df.describe().show()
```

```
+-------+------------------+------------------+-------------+-----------------+--------+
|summary|      binary_index|     coarse_labels|       coarse|      fine_labels|    fine|
+-------+------------------+------------------+-------------+-----------------+--------+
|  count|              4857|              4857|         4857|             4857|    4857|
|   mean|0.4968087296685197|14.012765081325922|         null| 60.6438130533251|    null|
| stddev|0.5000412946929902|2.0001651787719608|         null|15.04927242806568|    null|
|    min|               0.0|                12|medium_mammals|               34|     fox|
|    max|               1.0|                16| small_mammals|               80|squirrel|
+-------+------------------+------------------+-------------+-----------------+--------+

CPU times: user 9.18 ms, sys: 4.35 ms, total: 13.5 ms
Wall time: 1min 43s
```

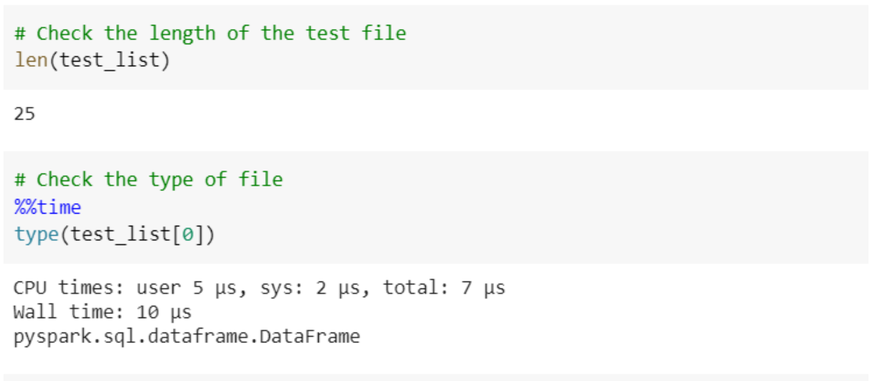
- Create a function for displaying the expected and predicted result.

```
# Function to print out randomly chosen images and their labels from predictions
%%time
def cifar_grid(predictions):
  samples = predictions.select(['coarse', 'prediction','fine','binary_index','features'])\
  .orderBy(rand()).limit(36).rdd.collect()

  fig = plt.figure(figsize=(15, 20))
  plt.subplots_adjust(hspace=2, wspace=2)
  index = 0
  n_row = 9
  n_col = 4
  for k in range(n_col):
    for j in range(n_row):
      #for sample in samples:
      i_inds = (j*n_col)+k
      label = ('small mammals' if samples[i_inds].binary_index == 0.0 else 'medium mammals')
      pred = ('small mammals' if samples[i_inds].prediction == 0.0 else 'medium mammals')
      ax= fig.add_subplot(9, 4, index+1, xticks=[], yticks=[])
      #ax = fig.add_subplot(n_row, n_col, i_inds+1)
      img = samples[i_inds].features.reshape(3,32,32).transpose([1, 2, 0])
      ax = plt.imshow(img, interpolation='nearest')
      if label == pred:
        plt.title("Predicted: {}\n Actual: {}".format(pred,label))
      else:
        plt.title("Predicted: {}\n Actual: {}".format(pred,label), color = "red")
      index += 1
  fig.set_tight_layout(True)
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 6.91 µs
```

5.1.2 Milestone 2

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



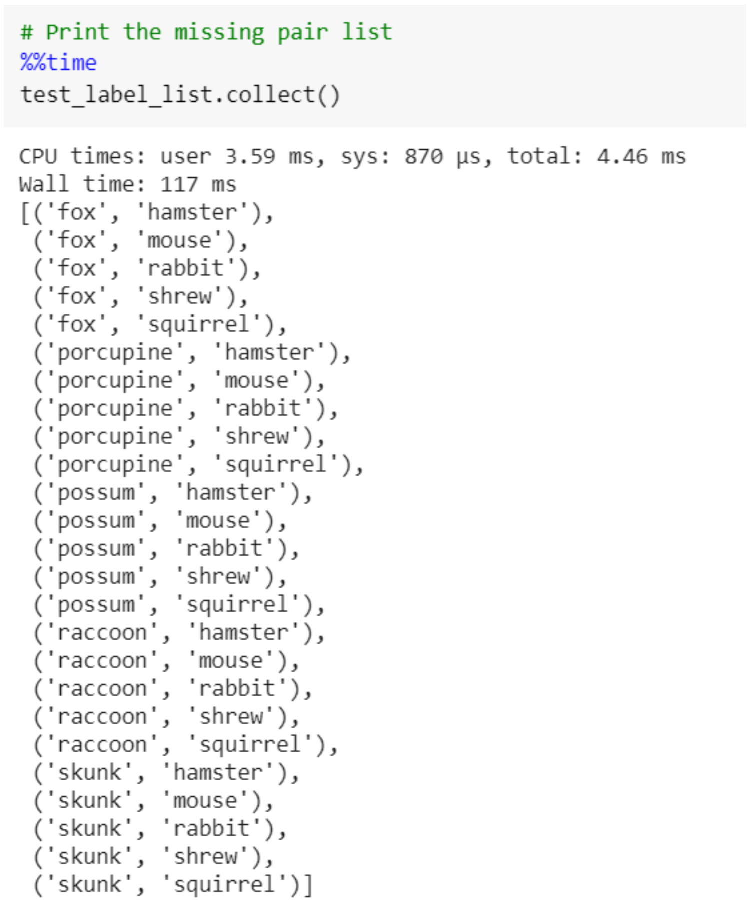
```
# Check the length of the test file
len(test_list)
```

```
25
```

```
# Check the type of file
%%time
type(test_list[0])
```

```
CPU times: user 5 µs, sys: 2 µs, total: 7 µs
Wall time: 10 µs
pyspark.sql.dataframe.DataFrame
```

- The test labels were checked and verified.



```
# Print the missing pair list
%%time
test_label_list.collect()
```

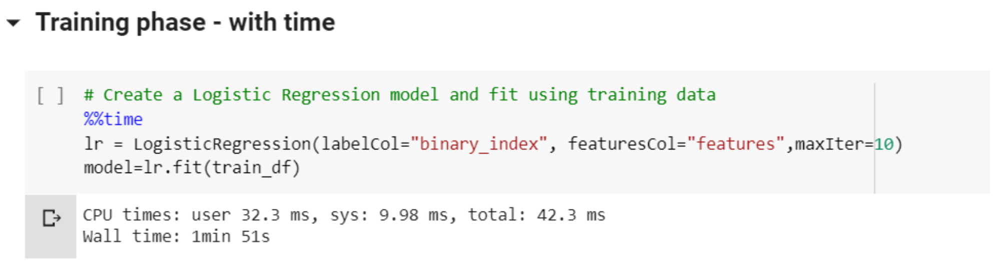
```
CPU times: user 3.59 ms, sys: 870 µs, total: 4.46 ms
Wall time: 117 ms
[('fox', 'hamster'),
 ('fox', 'mouse'),
 ('fox', 'rabbit'),
 ('fox', 'shrew'),
 ('fox', 'squirrel'),
 ('porcupine', 'hamster'),
 ('porcupine', 'mouse'),
 ('porcupine', 'rabbit'),
 ('porcupine', 'shrew'),
 ('porcupine', 'squirrel'),
 ('possum', 'hamster'),
 ('possum', 'mouse'),
 ('possum', 'rabbit'),
 ('possum', 'shrew'),
 ('possum', 'squirrel'),
 ('raccoon', 'hamster'),
 ('raccoon', 'mouse'),
 ('raccoon', 'rabbit'),
 ('raccoon', 'shrew'),
 ('raccoon', 'squirrel'),
 ('skunk', 'hamster'),
 ('skunk', 'mouse'),
 ('skunk', 'rabbit'),
 ('skunk', 'shrew'),
 ('skunk', 'squirrel')]
```

5.2 Model Creation

5.2.1 Milestone 1

5.2.1.1 Logistic Regression

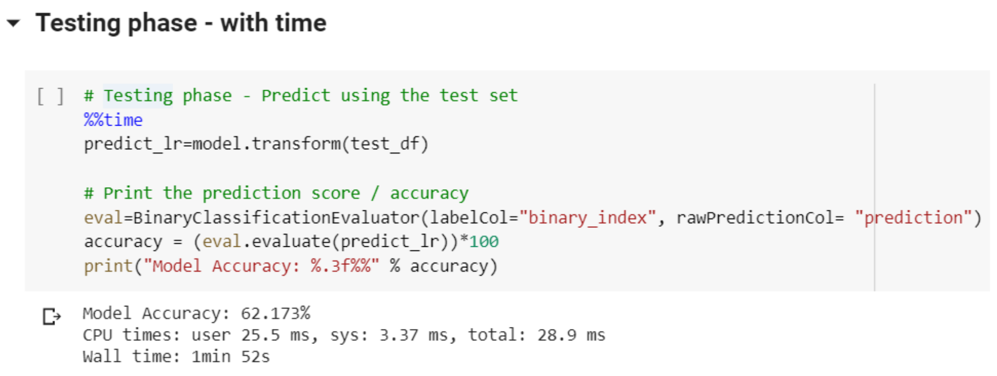
- Using ***LogisticRegression()*** function from pyspark ml library, we instantiate the model and fit the same using training data. Thereafter, we predict by transforming the testing data, and calculate the score using ***BinaryClassificationEvaluator()*** function.



Training phase - with time

```
[ ] # Create a Logistic Regression model and fit using training data
    %%time
    lr = LogisticRegression(labelCol="binary_index", featuresCol="features",maxIter=10)
    model=lr.fit(train_df)
```

```
CPU times: user 32.3 ms, sys: 9.98 ms, total: 42.3 ms
Wall time: 1min 51s
```

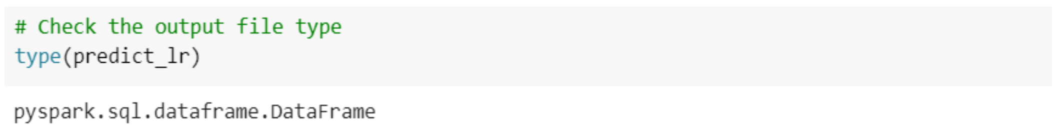


Testing phase - with time

```
[ ] # Testing phase - Predict using the test set
    %%time
    predict_lr=model.transform(test_df)

    # Print the prediction score / accuracy
    eval=BinaryClassificationEvaluator(labelCol="binary_index", rawPredictionCol= "prediction")
    accuracy = (eval.evaluate(predict_lr))*100
    print("Model Accuracy: %.3f%%" % accuracy)
```

```
Model Accuracy: 62.173%
CPU times: user 25.5 ms, sys: 3.37 ms, total: 28.9 ms
Wall time: 1min 52s
```

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



```
# Check the output file type
type(predict_lr)
```

```
pyspark.sql.dataframe.DataFrame
```

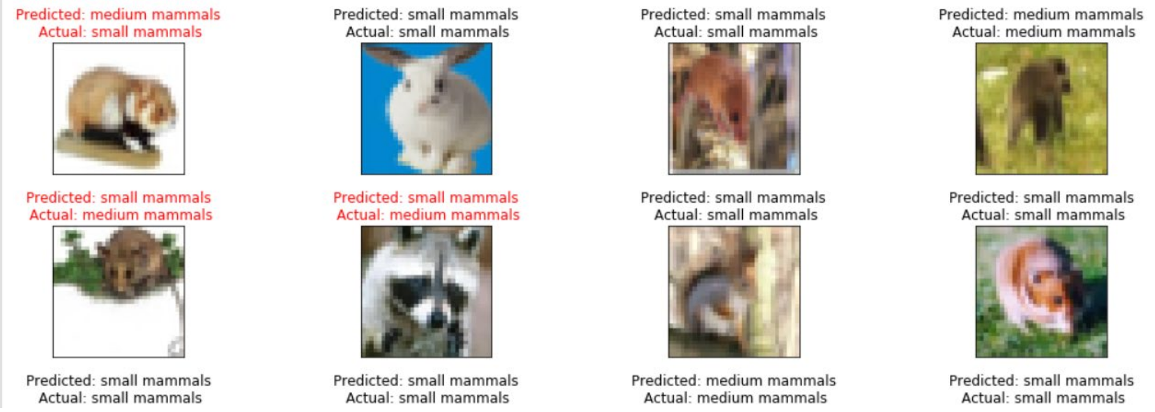
```
# Print the predictions
%%time
predict_lr.select("coarse","binary_index","prediction","probability").show(10)
```

```
+--------------+------------+----------+--------------------+
|        coarse|binary_index|prediction|         probability|
+--------------+------------+----------+--------------------+
| small_mammals|         0.0|       1.0|[0.47472145551301...|
|medium_mammals|         1.0|       1.0|[0.43350257031928...|
| small_mammals|         0.0|       0.0|[0.66759395480564...|
| small_mammals|         0.0|       0.0|[0.65729166310509...|
|medium_mammals|         1.0|       1.0|[0.43337212473262...|
|medium_mammals|         1.0|       0.0|[0.67365162847310...|
|medium_mammals|         1.0|       1.0|[0.40707934151075...|
|medium_mammals|         1.0|       0.0|[0.50087052162286...|
|medium_mammals|         1.0|       1.0|[0.14650473985964...|
|medium_mammals|         1.0|       1.0|[0.43103612241028...|
+--------------+------------+----------+--------------------+
only showing top 10 rows

CPU times: user 6.12 ms, sys: 3.06 ms, total: 9.18 ms
Wall time: 44.2 s
```

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

```
# Visualize 36 random images with original and predicted labels
%%time
cifar_grid(predict_lr)
```

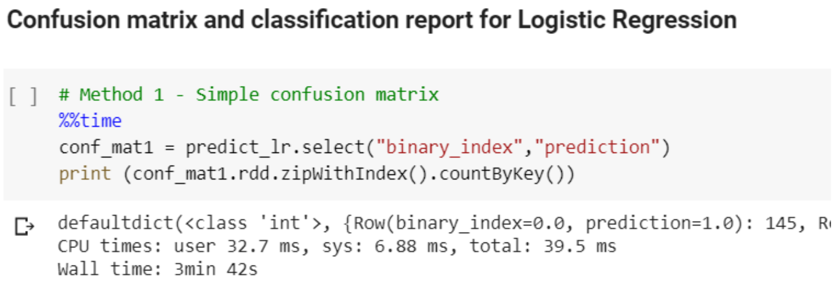
```
CPU times: user 655 ms, sys: 11.3 ms, total: 666 ms
Wall time: 1min 52s
```



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

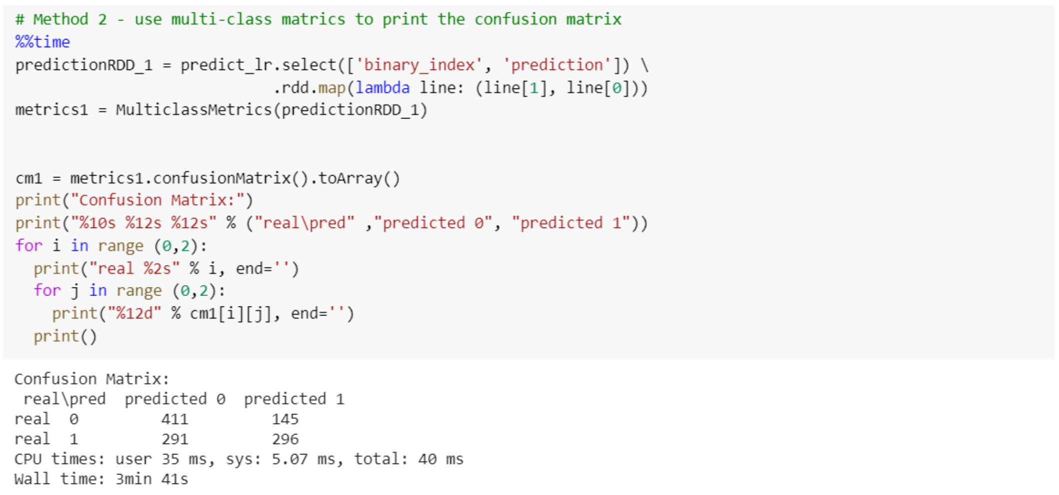
  Note: Comparison with Scikit-learn is provided in later sections.

  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.

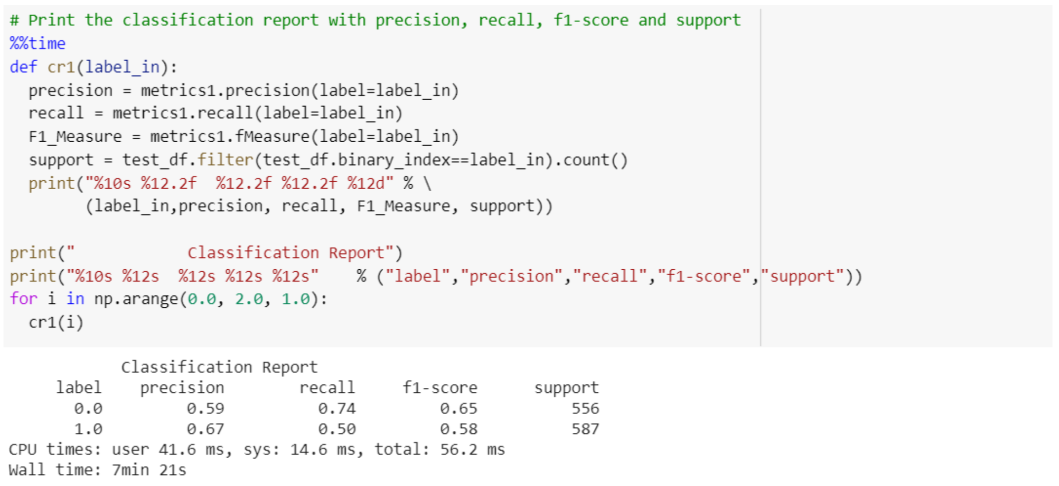


Confusion matrix and classification report for Logistic Regression

```
[ ] # Method 1 - Simple confusion matrix
    %%time
    conf_mat1 = predict_lr.select("binary_index","prediction")
    print (conf_mat1.rdd.zipWithIndex().countByKey())

    defaultdict(<class 'int'>, {Row(binary_index=0.0, prediction=1.0): 145, R
    CPU times: user 32.7 ms, sys: 6.88 ms, total: 39.5 ms
    Wall time: 3min 42s
```



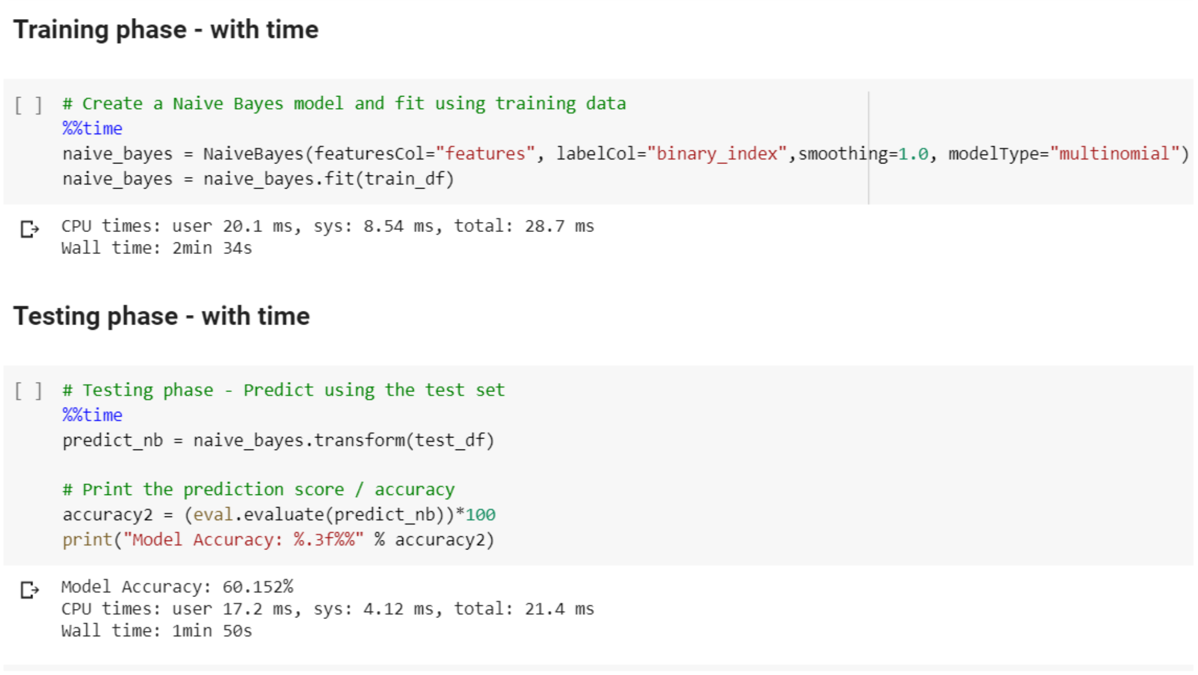
```
# Method 2 - use multi-class matrics to print the confusion matrix
%%time
predictionRDD_1 = predict_lr.select(['binary_index', 'prediction']) \
                            .rdd.map(lambda line: (line[1], line[0]))
metrics1 = MulticlassMetrics(predictionRDD_1)


cm1 = metrics1.confusionMatrix().toArray()
print("Confusion Matrix:")
print("%10s %12s %12s" % ("real\pred" ,"predicted 0", "predicted 1"))
for i in range (0,2):
  print("real %2s" % i, end='')
  for j in range (0,2):
    print("%12d" % cm1[i][j], end='')
  print()

Confusion Matrix:
 real\pred  predicted 0  predicted 1
real  0          411          145
real  1          291          296
CPU times: user 35 ms, sys: 5.07 ms, total: 40 ms
Wall time: 3min 41s
```



```
# Print the classification report with precision, recall, f1-score and support
%%time
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))

print("            Classification Report")
print("%10s %12s  %12s %12s %12s"    % ("label","precision","recall","f1-score","support"))
for i in np.arange(0.0, 2.0, 1.0):
  cr1(i)

            Classification Report
     label    precision         recall     f1-score      support
       0.0         0.59           0.74         0.65          556
       1.0         0.67           0.50         0.58          587
CPU times: user 41.6 ms, sys: 14.6 ms, total: 56.2 ms
Wall time: 7min 21s
```

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.



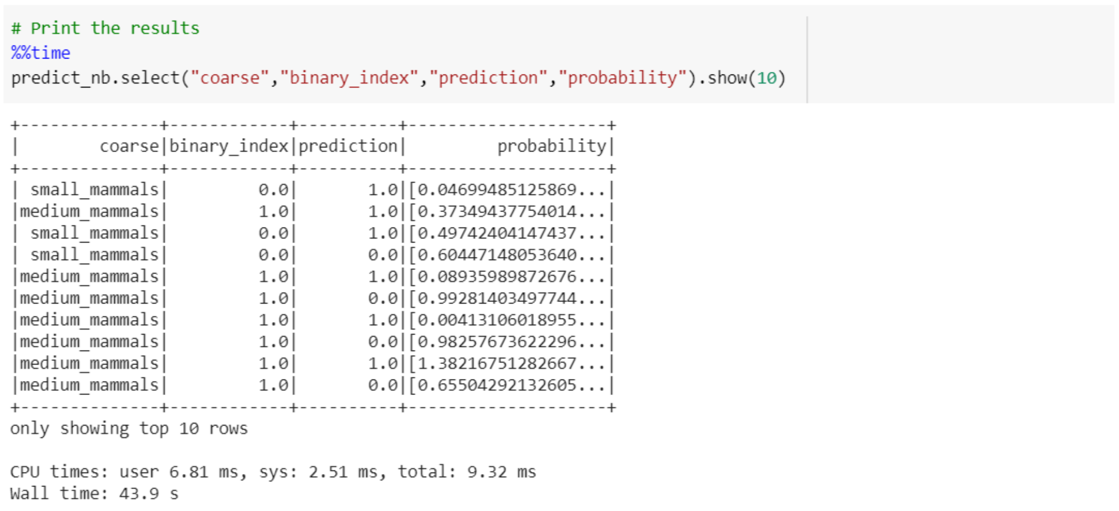
Training phase - with time

```
[ ] # Create a Naive Bayes model and fit using training data
    %%time
    naive_bayes = NaiveBayes(featuresCol="features", labelCol="binary_index",smoothing=1.0, modelType="multinomial")
    naive_bayes = naive_bayes.fit(train_df)

    CPU times: user 20.1 ms, sys: 8.54 ms, total: 28.7 ms
    Wall time: 2min 34s
```

Testing phase - with time

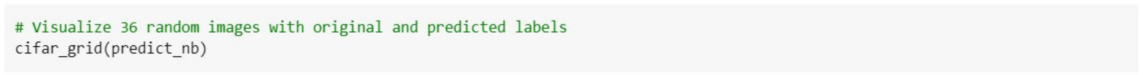
```
[ ] # Testing phase - Predict using the test set
    %%time
    predict_nb = naive_bayes.transform(test_df)

    # Print the prediction score / accuracy
    accuracy2 = (eval.evaluate(predict_nb))*100
    print("Model Accuracy: %.3f%%" % accuracy2)

    Model Accuracy: 60.152%
    CPU times: user 17.2 ms, sys: 4.12 ms, total: 21.4 ms
    Wall time: 1min 50s
```

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

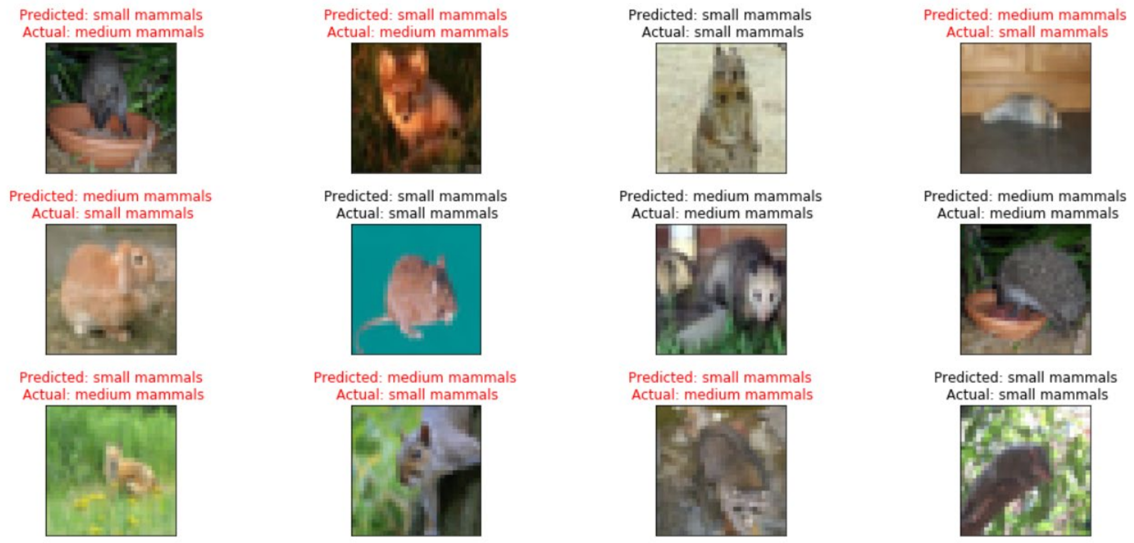


```
# Print the results
%%time
predict_nb.select("coarse","binary_index","prediction","probability").show(10)

+--------------+------------+----------+--------------------+
|        coarse|binary_index|prediction|         probability|
+--------------+------------+----------+--------------------+
| small_mammals|         0.0|       1.0|[0.04699485125869...|
|medium_mammals|         1.0|       1.0|[0.37349437754014...|
| small_mammals|         0.0|       1.0|[0.49742404147437...|
| small_mammals|         0.0|       0.0|[0.60447148053640...|
|medium_mammals|         1.0|       1.0|[0.08935989872676...|
|medium_mammals|         1.0|       0.0|[0.99281403497744...|
|medium_mammals|         1.0|       1.0|[0.00413106018955...|
|medium_mammals|         1.0|       0.0|[0.98257673622296...|
|medium_mammals|         1.0|       1.0|[1.38216751282667...|
|medium_mammals|         1.0|       0.0|[0.65504292132605...|
+--------------+------------+----------+--------------------+
only showing top 10 rows

CPU times: user 6.81 ms, sys: 2.51 ms, total: 9.32 ms
Wall time: 43.9 s
```

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



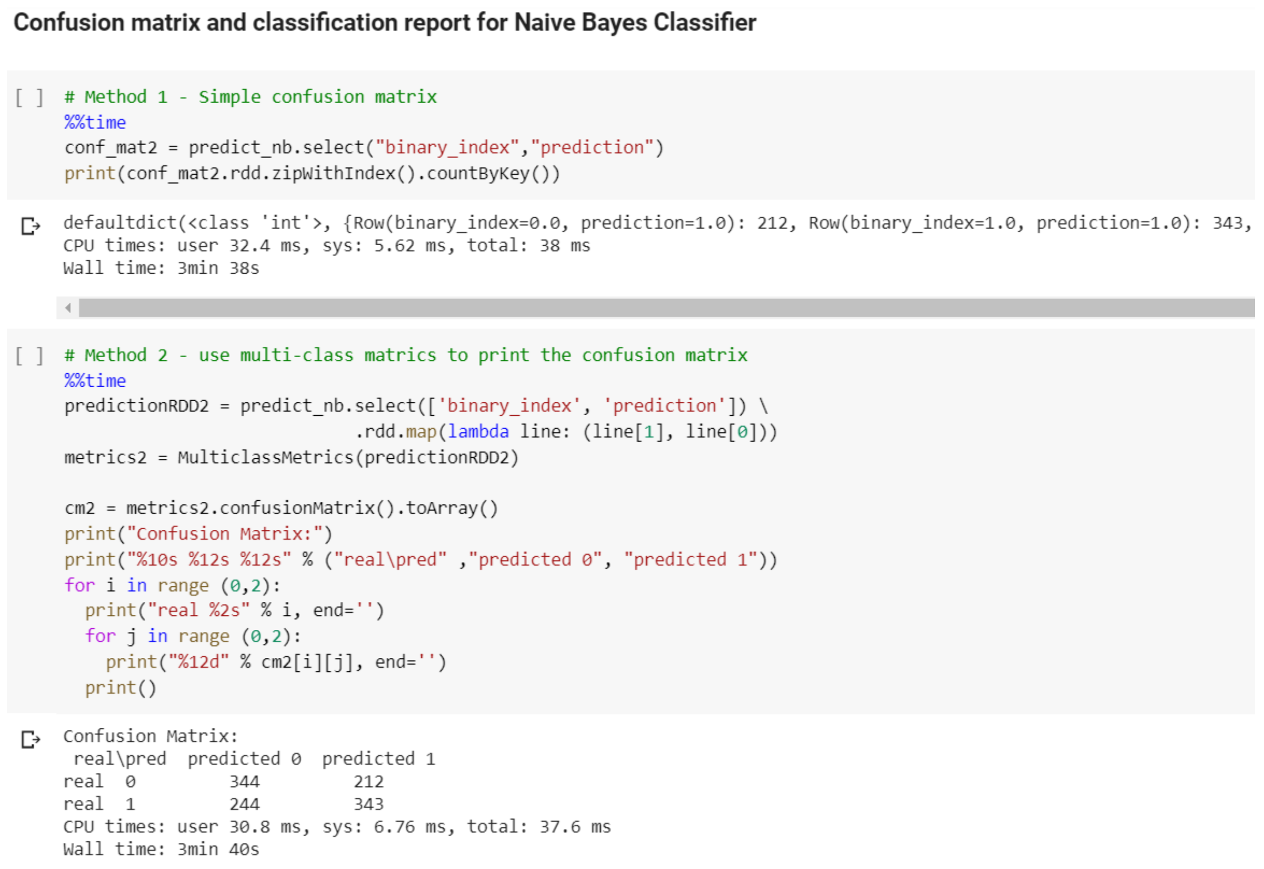
```
# Visualize 36 random images with original and predicted labels
cifar_grid(predict_nb)
```



- 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.



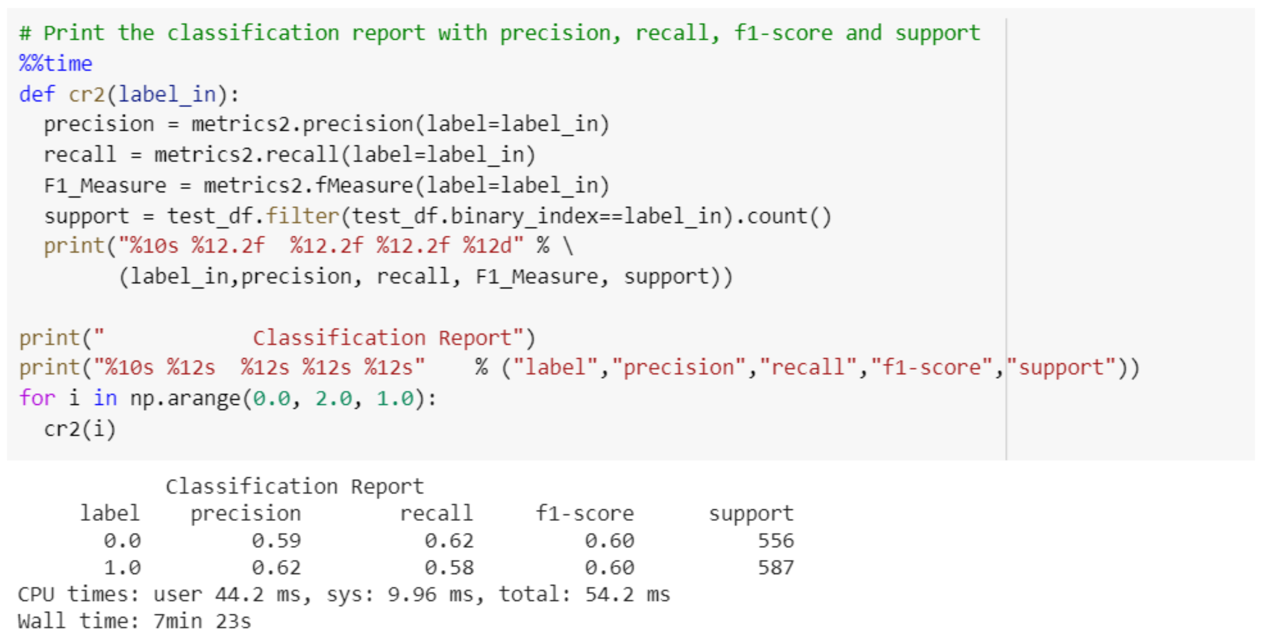
Confusion matrix and classification report for Naive Bayes Classifier

```
[ ] # Method 1 - Simple confusion matrix
    %%time
    conf_mat2 = predict_nb.select("binary_index","prediction")
    print(conf_mat2.rdd.zipWithIndex().countByKey())
```

```
defaultdict(<class 'int'>, {Row(binary_index=0.0, prediction=1.0): 212, Row(binary_index=1.0, prediction=1.0): 343,
CPU times: user 32.4 ms, sys: 5.62 ms, total: 38 ms
Wall time: 3min 38s
```

```
[ ] # Method 2 - use multi-class matrics to print the confusion matrix
    %%time
    predictionRDD2 = predict_nb.select(['binary_index', 'prediction']) \
                                .rdd.map(lambda line: (line[1], line[0]))
    metrics2 = MulticlassMetrics(predictionRDD2)

    cm2 = metrics2.confusionMatrix().toArray()
    print("Confusion Matrix:")
    print("%10s %12s %12s" % ("real\pred" ,"predicted 0", "predicted 1"))
    for i in range (0,2):
      print("real %2s" % i, end='')
      for j in range (0,2):
        print("%12d" % cm2[i][j], end='')
      print()
```

```
Confusion Matrix:
 real\pred  predicted 0  predicted 1
real  0         344         212
real  1         244         343
CPU times: user 30.8 ms, sys: 6.76 ms, total: 37.6 ms
Wall time: 3min 40s
```



```
# Print the classification report with precision, recall, f1-score and support
%%time
def cr2(label_in):
  precision = metrics2.precision(label=label_in)
  recall = metrics2.recall(label=label_in)
  F1_Measure = metrics2.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))

print("            Classification Report")
print("%10s %12s  %12s %12s %12s"    % ("label","precision","recall","f1-score","support"))
for i in np.arange(0.0, 2.0, 1.0):
  cr2(i)
```

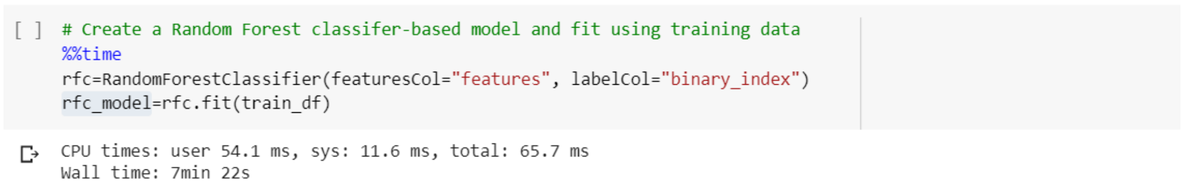
```
            Classification Report
     label    precision        recall     f1-score      support
       0.0         0.59          0.62         0.60          556
       1.0         0.62          0.58         0.60          587
CPU times: user 44.2 ms, sys: 9.96 ms, total: 54.2 ms
Wall time: 7min 23s
```

Please refer subsequent sections for comparison with Scikit-learn.

5.1.1.3 Random Forest

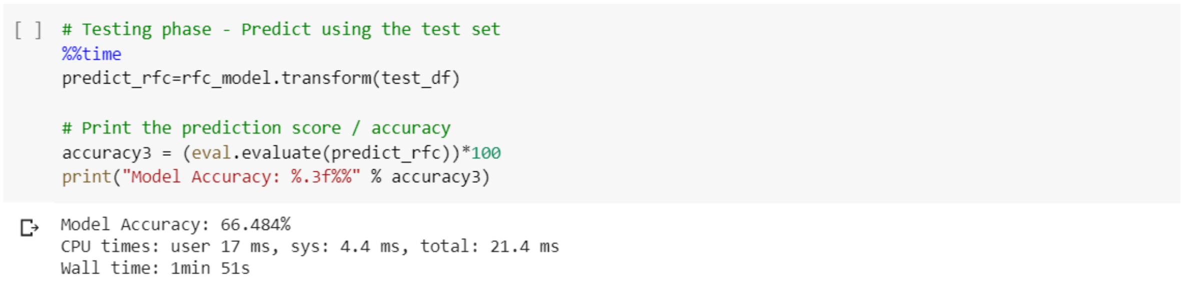
- Using ***RandomForestClassifier()*** function, we instantiate the model and fit the same using training data. Thereafter, we predict by transforming the testing data, and calculate the score using ***BinaryClassificationEvaluator()*** function.

Training phase - with time



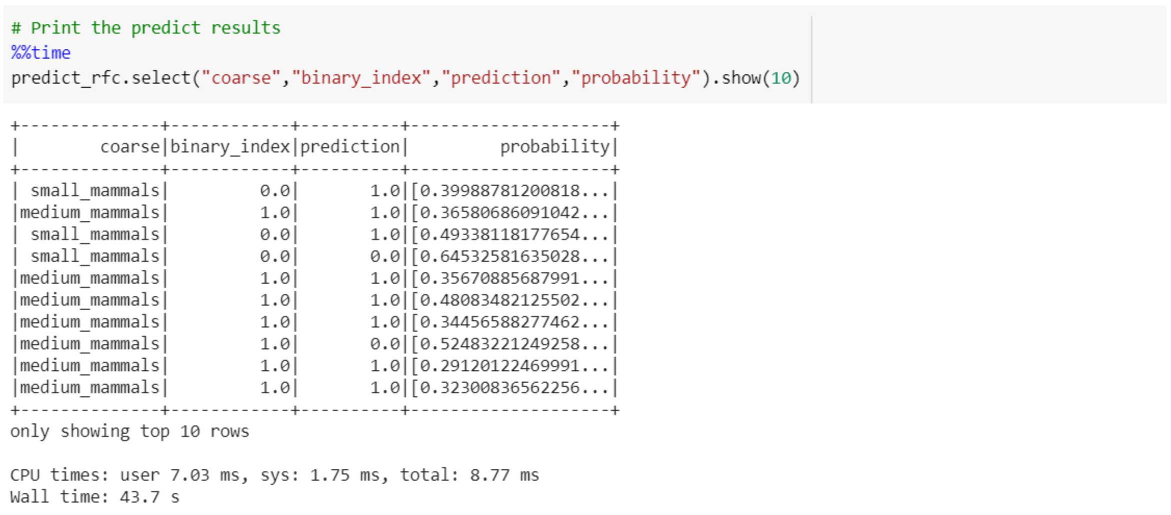
```
[ ] # Create a Random Forest classifer-based model and fit using training data
    %%time
    rfc=RandomForestClassifier(featuresCol="features", labelCol="binary_index")
    rfc_model=rfc.fit(train_df)

    CPU times: user 54.1 ms, sys: 11.6 ms, total: 65.7 ms
    Wall time: 7min 22s
```

Testing phase - with time



```
[ ] # Testing phase - Predict using the test set
    %%time
    predict_rfc=rfc_model.transform(test_df)

    # Print the prediction score / accuracy
    accuracy3 = (eval.evaluate(predict_rfc))*100
    print("Model Accuracy: %.3f%%" % accuracy3)

    Model Accuracy: 66.484%
    CPU times: user 17 ms, sys: 4.4 ms, total: 21.4 ms
    Wall time: 1min 51s
```

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



```
# Print the predict results
%%time
predict_rfc.select("coarse","binary_index","prediction","probability").show(10)

+--------------+------------+----------+--------------------+
|        coarse|binary_index|prediction|         probability|
+--------------+------------+----------+--------------------+
| small_mammals|         0.0|       1.0|[0.39988781200818...|
|medium_mammals|         1.0|       1.0|[0.36580686091042...|
| small_mammals|         0.0|       1.0|[0.49338118177654...|
| small_mammals|         0.0|       0.0|[0.64532581635028...|
|medium_mammals|         1.0|       1.0|[0.35670885687991...|
|medium_mammals|         1.0|       1.0|[0.48083482125502...|
|medium_mammals|         1.0|       1.0|[0.34456588277462...|
|medium_mammals|         1.0|       0.0|[0.52483221249258...|
|medium_mammals|         1.0|       1.0|[0.29120122469991...|
|medium_mammals|         1.0|       1.0|[0.32300836562256...|
+--------------+------------+----------+--------------------+
only showing top 10 rows

CPU times: user 7.03 ms, sys: 1.75 ms, total: 8.77 ms
Wall time: 43.7 s
```

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

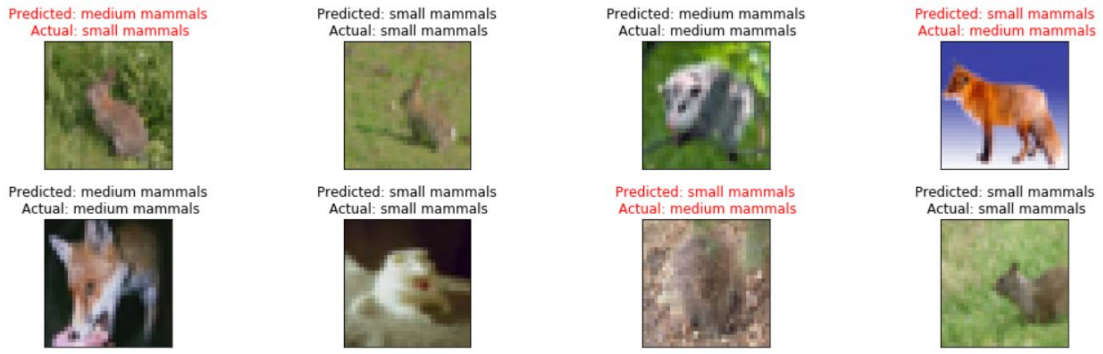
```
# Check the parameter map
rfc_model.extractParamMap()
```

```
{Param(parent='RandomForestClassifier_09a79c2346ad', name='cacheNodeIds', doc='If false, the algorithm will pass tr
 Param(parent='RandomForestClassifier_09a79c2346ad', name='checkpointInterval', doc='set checkpoint interval (>= 1)
 Param(parent='RandomForestClassifier_09a79c2346ad', name='featureSubsetStrategy', doc='The number of features to c
 Param(parent='RandomForestClassifier_09a79c2346ad', name='featuresCol', doc='features column name'): 'features',
 Param(parent='RandomForestClassifier_09a79c2346ad', name='impurity', doc='Criterion used for information gain calc
 Param(parent='RandomForestClassifier_09a79c2346ad', name='labelCol', doc='label column name'): 'binary_index',
 Param(parent='RandomForestClassifier_09a79c2346ad', name='maxBins', doc='Max number of bins for discretizing conti
 Param(parent='RandomForestClassifier_09a79c2346ad', name='maxDepth', doc='Maximum depth of the tree. (>= 0) E.g.,
 Param(parent='RandomForestClassifier_09a79c2346ad', name='maxMemoryInMB', doc='Maximum memory in MB allocated to h
 Param(parent='RandomForestClassifier_09a79c2346ad', name='minInfoGain', doc='Minimum information gain for a split
 Param(parent='RandomForestClassifier_09a79c2346ad', name='minInstancesPerNode', doc='Minimum number of instances e
 Param(parent='RandomForestClassifier_09a79c2346ad', name='numTrees', doc='Number of trees to train (>= 1)'): 20,
 Param(parent='RandomForestClassifier_09a79c2346ad', name='predictionCol', doc='prediction column name'): 'predicti
 Param(parent='RandomForestClassifier_09a79c2346ad', name='probabilityCol', doc='Column name for predicted class co
 Param(parent='RandomForestClassifier_09a79c2346ad', name='rawPredictionCol', doc='raw prediction (a.k.a. confidenc
 Param(parent='RandomForestClassifier_09a79c2346ad', name='seed', doc='random seed'): -110140864341450282,
 Param(parent='RandomForestClassifier_09a79c2346ad', name='subsamplingRate', doc='Fraction of the training data use
```

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

```
# Visualize 36 random images with original and predicted labels
%%time
cifar_grid(predict_rfc)
```

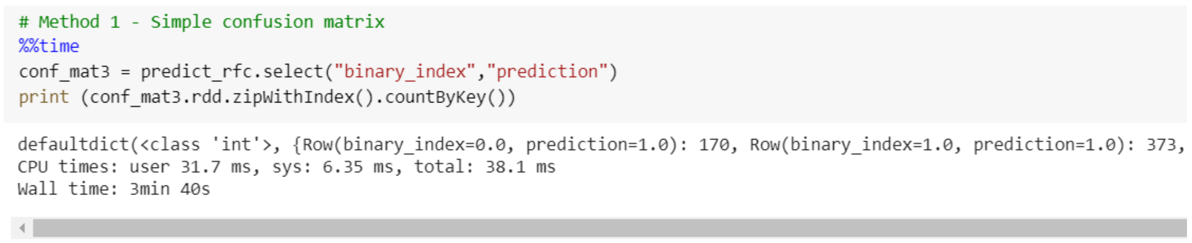
```
CPU times: user 661 ms, sys: 517 µs, total: 662 ms
Wall time: 1min 50s
```



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

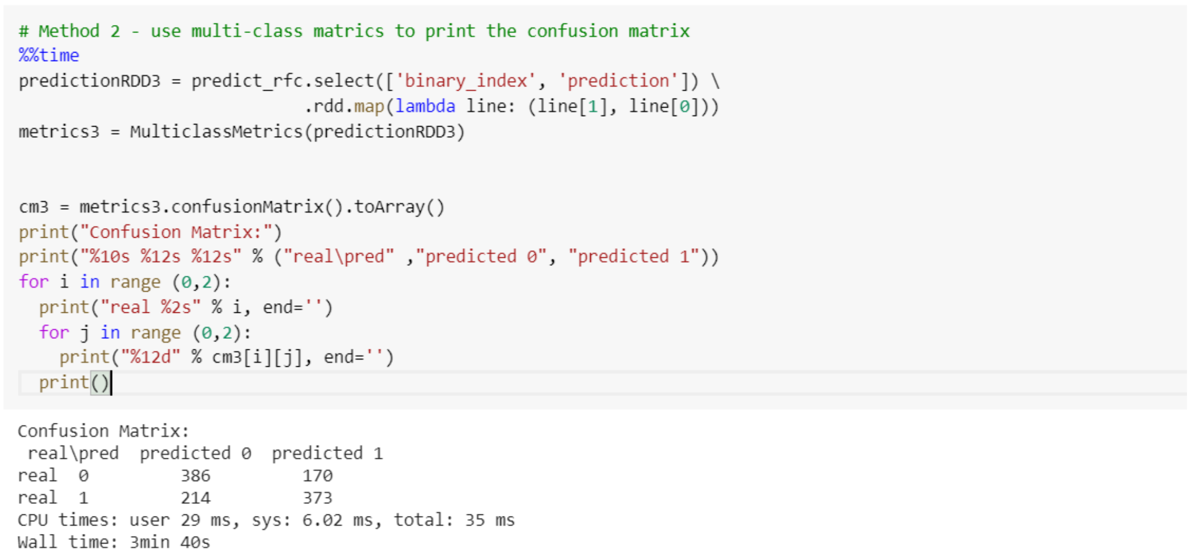
  Note: Comparison with Scikit-learn is provided in later sections.

  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.



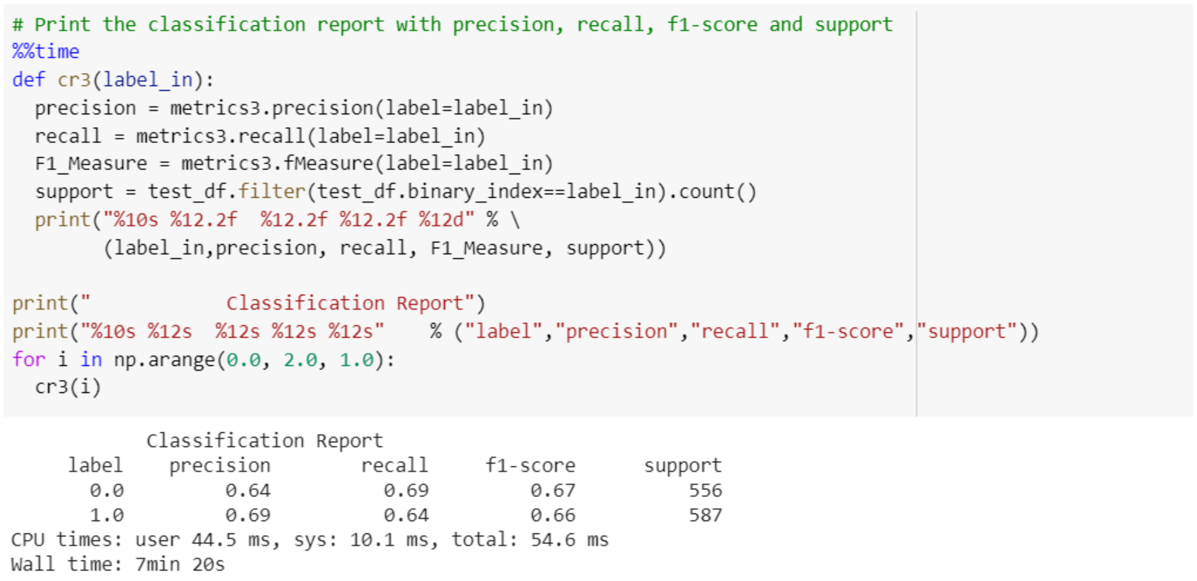
```
# Method 1 - Simple confusion matrix
%%time
conf_mat3 = predict_rfc.select("binary_index","prediction")
print (conf_mat3.rdd.zipWithIndex().countByKey())
```

```
defaultdict(<class 'int'>, {Row(binary_index=0.0, prediction=1.0): 170, Row(binary_index=1.0, prediction=1.0): 373,
CPU times: user 31.7 ms, sys: 6.35 ms, total: 38.1 ms
Wall time: 3min 40s
```



```
# Method 2 - use multi-class matrics to print the confusion matrix
%%time
predictionRDD3 = predict_rfc.select(['binary_index', 'prediction']) \
                            .rdd.map(lambda line: (line[1], line[0]))
metrics3 = MulticlassMetrics(predictionRDD3)


cm3 = metrics3.confusionMatrix().toArray()
print("Confusion Matrix:")
print("%10s %12s %12s" % ("real\pred" ,"predicted 0", "predicted 1"))
for i in range (0,2):
  print("real %2s" % i, end='')
  for j in range (0,2):
    print("%12d" % cm3[i][j], end='')
  print()
```

```
Confusion Matrix:
 real\pred  predicted 0  predicted 1
real  0         386          170
real  1         214          373
CPU times: user 29 ms, sys: 6.02 ms, total: 35 ms
Wall time: 3min 40s
```



```
# Print the classification report with precision, recall, f1-score and support
%%time
def cr3(label_in):
  precision = metrics3.precision(label=label_in)
  recall = metrics3.recall(label=label_in)
  F1_Measure = metrics3.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))

print("            Classification Report")
print("%10s %12s  %12s %12s %12s"    % ("label","precision","recall","f1-score","support"))
for i in np.arange(0.0, 2.0, 1.0):
  cr3(i)
```

```
            Classification Report
     label    precision        recall     f1-score       support
       0.0         0.64          0.69         0.67           556
       1.0         0.69          0.64         0.66           587
CPU times: user 44.5 ms, sys: 10.1 ms, total: 54.6 ms
Wall time: 7min 20s
```

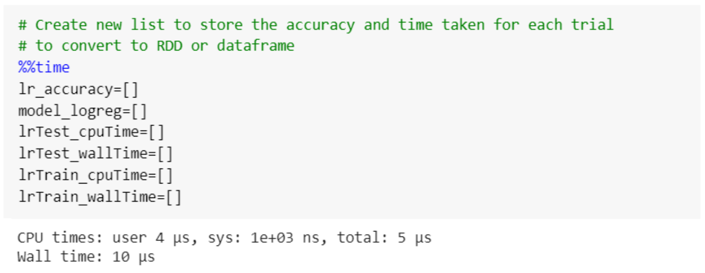
Please refer the subsequent sections for comparison with Scikit-learn.

5.2.2 Milestone 2:

We chose the best 2 models with superior accuracy based on Milestone 1.

5.2.2.1 Logistic Regression

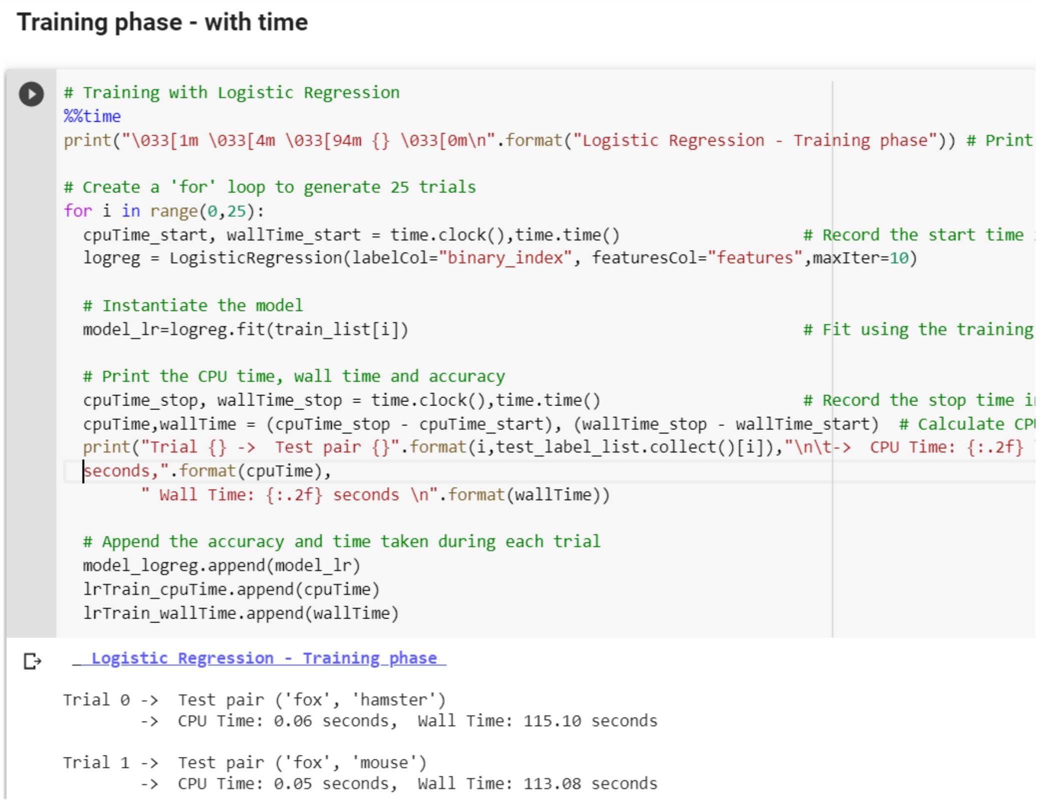
- Empty lists were initialized to store the model inputs. They were thereafter converted to Pyspark dataframes.



```
# Create new list to store the accuracy and time taken for each trial
# to convert to RDD or dataframe
%%time
lr_accuracy=[]
model_logreg=[]
lrTest_cpuTime=[]
lrTest_wallTime=[]
lrTrain_cpuTime=[]
lrTrain_wallTime=[]
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10 µs
```

- 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.

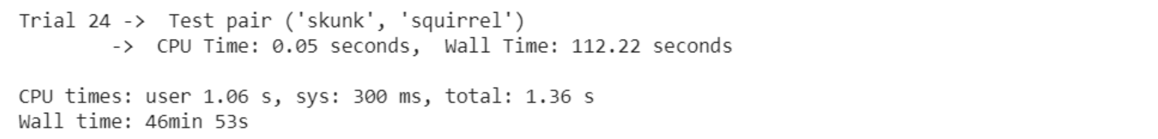


Training phase - with time

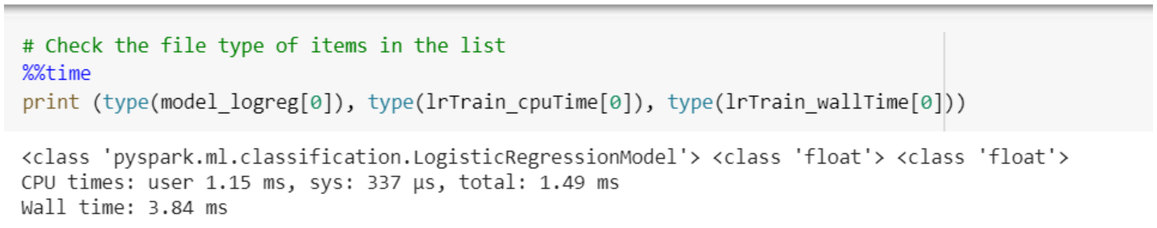
```
# Training with Logistic Regression
%%time
print("\033[1m \033[4m \033[94m {} \033[0m\n".format("Logistic Regression - Training phase")) # Print

# Create a 'for' loop to generate 25 trials
for i in range(0,25):
  cpuTime_start, wallTime_start = time.clock(),time.time()                    # Record the start time
  logreg = LogisticRegression(labelCol="binary_index", featuresCol="features",maxIter=10)

  # Instantiate the model
  model_lr=logreg.fit(train_list[i])                                          # Fit using the training

  # Print the CPU time, wall time and accuracy
  cpuTime_stop, wallTime_stop = time.clock(),time.time()                      # Record the stop time i
  cpuTime,wallTime = (cpuTime_stop - cpuTime_start), (wallTime_stop - wallTime_start)  # Calculate CP
  print("Trial {} ->  Test pair {}".format(i,test_label_list.collect()[i]),"\n\t->  CPU Time: {:.2f}
  seconds,".format(cpuTime),
        " Wall Time: {:.2f} seconds \n".format(wallTime))

  # Append the accuracy and time taken during each trial
  model_logreg.append(model_lr)
  lrTrain_cpuTime.append(cpuTime)
  lrTrain_wallTime.append(wallTime)
```

```
 Logistic Regression - Training phase 

Trial 0 ->  Test pair ('fox', 'hamster')
        ->  CPU Time: 0.06 seconds,  Wall Time: 115.10 seconds

Trial 1 ->  Test pair ('fox', 'mouse')
        ->  CPU Time: 0.05 seconds,  Wall Time: 113.08 seconds
```

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



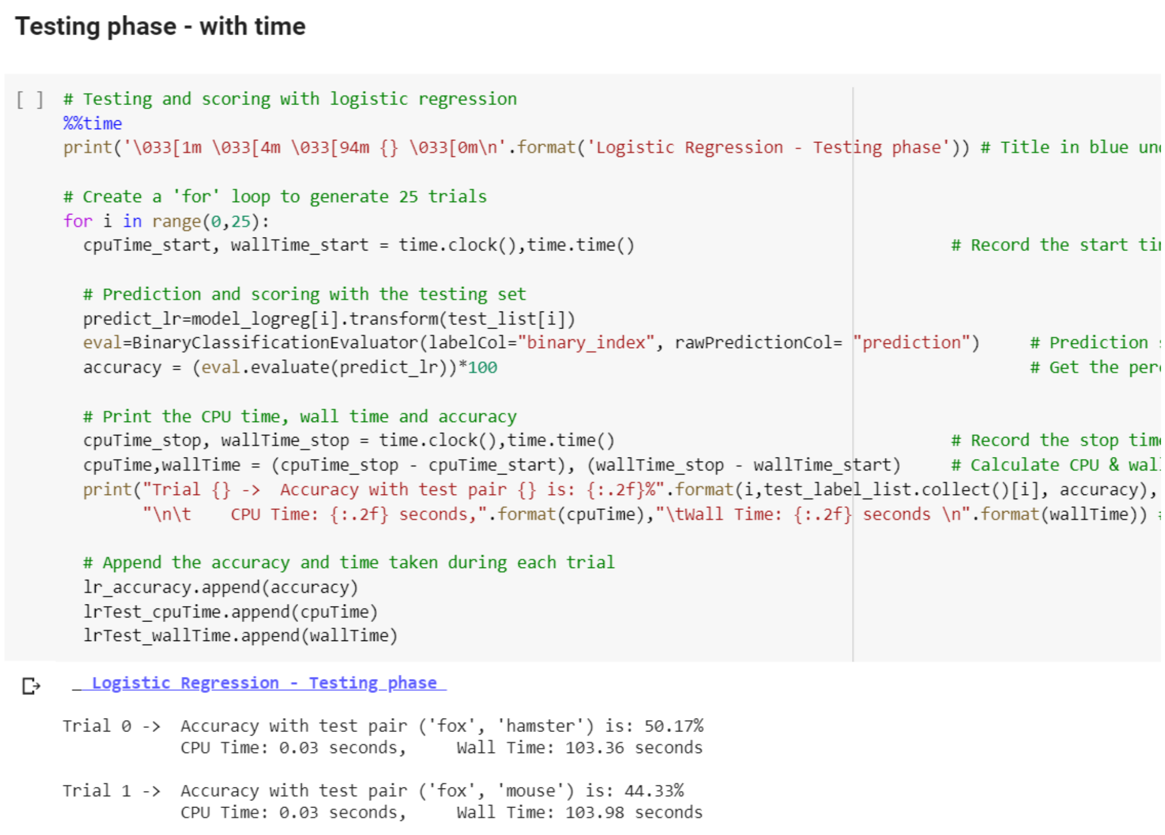
```
Trial 24 ->  Test pair ('skunk', 'squirrel')
        ->  CPU Time: 0.05 seconds,  Wall Time: 112.22 seconds

CPU times: user 1.06 s, sys: 300 ms, total: 1.36 s
Wall time: 46min 53s
```



```
# Check the file type of items in the list
%%time
print (type(model_logreg[0]), type(lrTrain_cpuTime[0]), type(lrTrain_wallTime[0]))
```

```
<class 'pyspark.ml.classification.LogisticRegressionModel'> <class 'float'> <class 'float'>
CPU times: user 1.15 ms, sys: 337 µs, total: 1.49 ms
Wall time: 3.84 ms
```

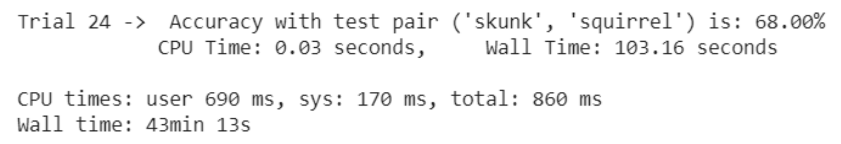
- Similarly, testing was performed.



Testing phase - with time

```
[ ] # Testing and scoring with logistic regression
    %%time
    print('\033[1m \033[4m \033[94m {} \033[0m\n'.format('Logistic Regression - Testing phase')) # Title in blue un

    # Create a 'for' loop to generate 25 trials
    for i in range(0,25):
      cpuTime_start, wallTime_start = time.clock(),time.time()                           # Record the start ti

      # Prediction and scoring with the testing set
      predict_lr=model_logreg[i].transform(test_list[i])
      eval=BinaryClassificationEvaluator(labelCol="binary_index", rawPredictionCol= "prediction")     # Prediction
      accuracy = (eval.evaluate(predict_lr))*100                                                      # Get the per

      # Print the CPU time, wall time and accuracy
      cpuTime_stop, wallTime_stop = time.clock(),time.time()                             # Record the stop time
      cpuTime,wallTime = (cpuTime_stop - cpuTime_start), (wallTime_stop - wallTime_start)     # Calculate CPU & wal
      print("Trial {} ->  Accuracy with test pair {} is: {:.2f}%".format(i,test_label_list.collect()[i], accuracy),
            "\n\t    CPU Time: {:.2f} seconds,".format(cpuTime),"\tWall Time: {:.2f} seconds \n".format(wallTime))

      # Append the accuracy and time taken during each trial
      lr_accuracy.append(accuracy)
      lrTest_cpuTime.append(cpuTime)
      lrTest_wallTime.append(wallTime)
```

```
  Logistic Regression - Testing phase

Trial 0 ->  Accuracy with test pair ('fox', 'hamster') is: 50.17%
            CPU Time: 0.03 seconds,     Wall Time: 103.36 seconds

Trial 1 ->  Accuracy with test pair ('fox', 'mouse') is: 44.33%
            CPU Time: 0.03 seconds,     Wall Time: 103.98 seconds
```

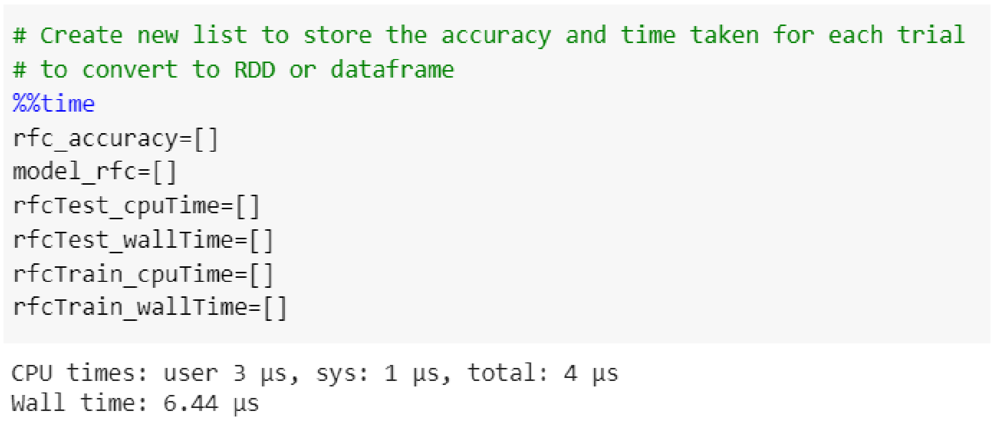


```
Trial 24 ->  Accuracy with test pair ('skunk', 'squirrel') is: 68.00%
            CPU Time: 0.03 seconds,     Wall Time: 103.16 seconds

CPU times: user 690 ms, sys: 170 ms, total: 860 ms
Wall time: 43min 13s
```

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

5.2.2.2 Random Forest

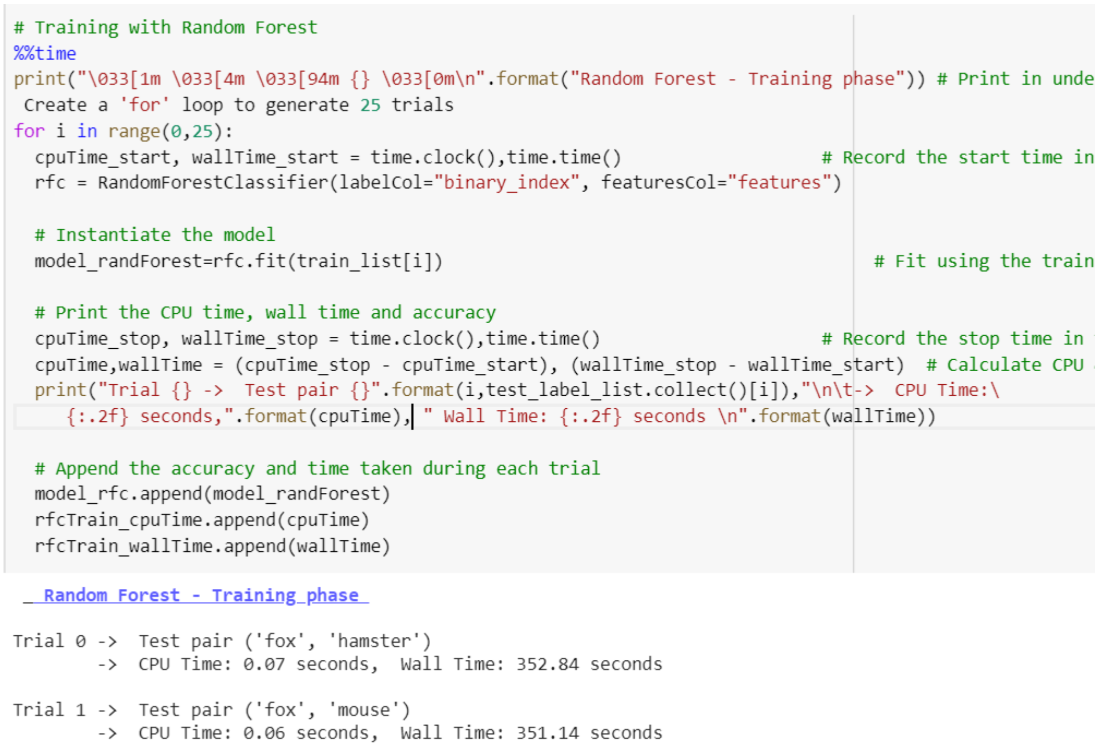
- Empty lists were initialized to store the model inputs. They were thereafter converted to Pyspark dataframes, once the lists were populated.



```
# Create new list to store the accuracy and time taken for each trial
# to convert to RDD or dataframe
%%time
rfc_accuracy=[]
model_rfc=[]
rfcTest_cpuTime=[]
rfcTest_wallTime=[]
rfcTrain_cpuTime=[]
rfcTrain_wallTime=[]
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 6.44 µs
```

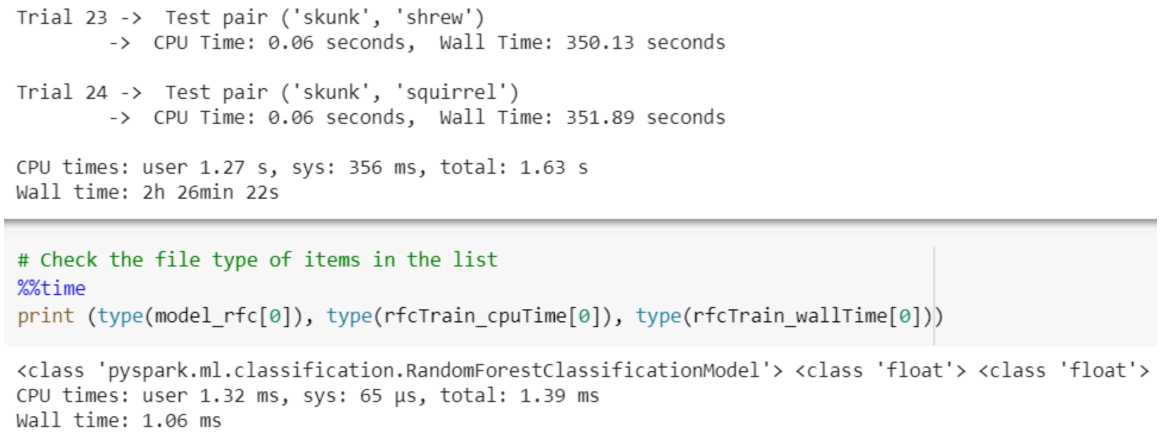
- Training was performed using the same function from milestone 1.
- However, we recorded the time using ***time.clock()*** for CPU time and ***time.time()*** for Wall time. We had to create a for loop to execute the training for 25 pairs and store results in a list.



```
# Training with Random Forest
%%time
print("\033[1m \033[4m \033[94m {} \033[0m\n".format("Random Forest - Training phase")) # Print in unde
 Create a 'for' loop to generate 25 trials
for i in range(0,25):
  cpuTime_start, wallTime_start = time.clock(),time.time()                   # Record the start time in
  rfc = RandomForestClassifier(labelCol="binary_index", featuresCol="features")

  # Instantiate the model
  model_randForest=rfc.fit(train_list[i])                                        # Fit using the train

  # Print the CPU time, wall time and accuracy
  cpuTime_stop, wallTime_stop = time.clock(),time.time()                     # Record the stop time in
  cpuTime,wallTime = (cpuTime_stop - cpuTime_start), (wallTime_stop - wallTime_start)  # Calculate CPU
  print("Trial {} ->  Test pair {}".format(i,test_label_list.collect()[i]),"\n\t->  CPU Time:\
     {:.2f} seconds,".format(cpuTime), " Wall Time: {:.2f} seconds \n".format(wallTime))

  # Append the accuracy and time taken during each trial
  model_rfc.append(model_randForest)
  rfcTrain_cpuTime.append(cpuTime)
  rfcTrain_wallTime.append(wallTime)
```

```
  Random Forest - Training phase 

Trial 0 ->  Test pair ('fox', 'hamster') 
        ->  CPU Time: 0.07 seconds,  Wall Time: 352.84 seconds 

Trial 1 ->  Test pair ('fox', 'mouse') 
        ->  CPU Time: 0.06 seconds,  Wall Time: 351.14 seconds 
```

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



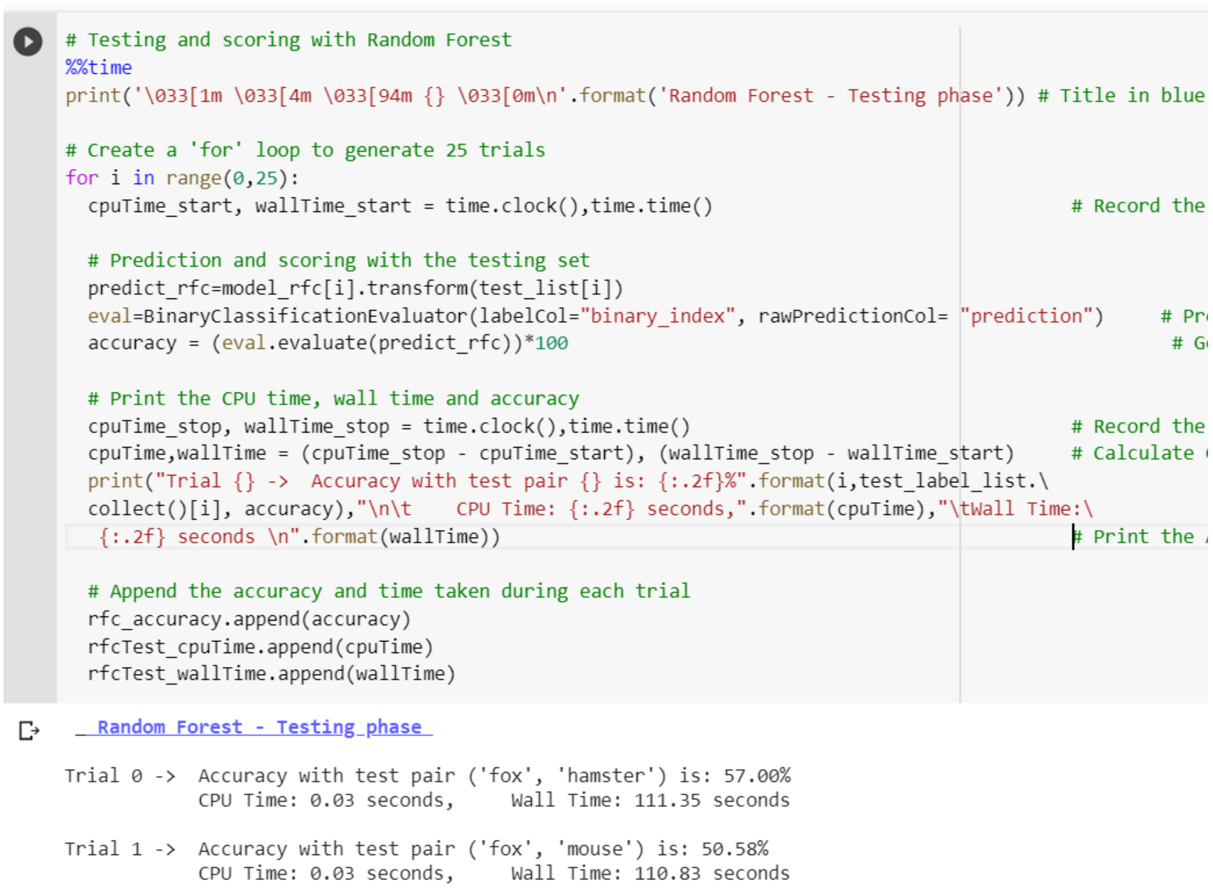
```
Trial 23 ->  Test pair ('skunk', 'shrew') 
         ->  CPU Time: 0.06 seconds,  Wall Time: 350.13 seconds 

Trial 24 ->  Test pair ('skunk', 'squirrel') 
         ->  CPU Time: 0.06 seconds,  Wall Time: 351.89 seconds 

CPU times: user 1.27 s, sys: 356 ms, total: 1.63 s
Wall time: 2h 26min 22s
```

```
# Check the file type of items in the list
%%time
print (type(model_rfc[0]), type(rfcTrain_cpuTime[0]), type(rfcTrain_wallTime[0]))
```

```
<class 'pyspark.ml.classification.RandomForestClassificationModel'> <class 'float'> <class 'float'>
CPU times: user 1.32 ms, sys: 65 µs, total: 1.39 ms
Wall time: 1.06 ms
```

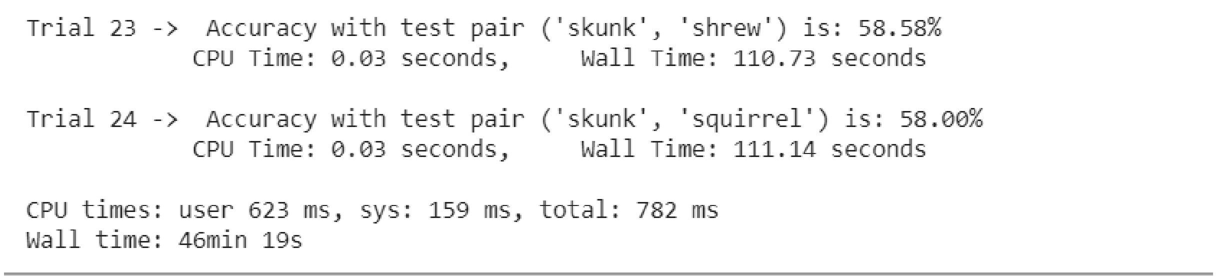
- Similarly, testing was performed.

Testing phase - with time



```
# Testing and scoring with Random Forest
%%time
print('\033[1m \033[4m \033[94m {} \033[0m\n'.format('Random Forest - Testing phase')) # Title in blue

# Create a 'for' loop to generate 25 trials
for i in range(0,25):
  cpuTime_start, wallTime_start = time.clock(),time.time()                                # Record the

  # Prediction and scoring with the testing set
  predict_rfc=model_rfc[i].transform(test_list[i])
  eval=BinaryClassificationEvaluator(labelCol="binary_index", rawPredictionCol= "prediction")     # Pr
  accuracy = (eval.evaluate(predict_rfc))*100                                                       # G

  # Print the CPU time, wall time and accuracy
  cpuTime_stop, wallTime_stop = time.clock(),time.time()                                  # Record the
  cpuTime,wallTime = (cpuTime_stop - cpuTime_start), (wallTime_stop - wallTime_start)     # Calculate
  print("Trial {} ->  Accuracy with test pair {} is: {:.2f}%".format(i,test_label_list.\
  collect()[i], accuracy),"\n\t    CPU Time: {:.2f} seconds,".format(cpuTime),"\tWall Time:\
   {:.2f} seconds \n".format(wallTime))                                                   # Print the

  # Append the accuracy and time taken during each trial
  rfc_accuracy.append(accuracy)
  rfcTest_cpuTime.append(cpuTime)
  rfcTest_wallTime.append(wallTime)
```

```
  Random Forest - Testing phase 

Trial 0 ->  Accuracy with test pair ('fox', 'hamster') is: 57.00%
            CPU Time: 0.03 seconds,     Wall Time: 111.35 seconds

Trial 1 ->  Accuracy with test pair ('fox', 'mouse') is: 50.58%
            CPU Time: 0.03 seconds,     Wall Time: 110.83 seconds
```



```
Trial 23 ->  Accuracy with test pair ('skunk', 'shrew') is: 58.58%
            CPU Time: 0.03 seconds,     Wall Time: 110.73 seconds

Trial 24 ->  Accuracy with test pair ('skunk', 'squirrel') is: 58.00%
            CPU Time: 0.03 seconds,     Wall Time: 111.14 seconds

CPU times: user 623 ms, sys: 159 ms, total: 782 ms
Wall time: 46min 19s
```

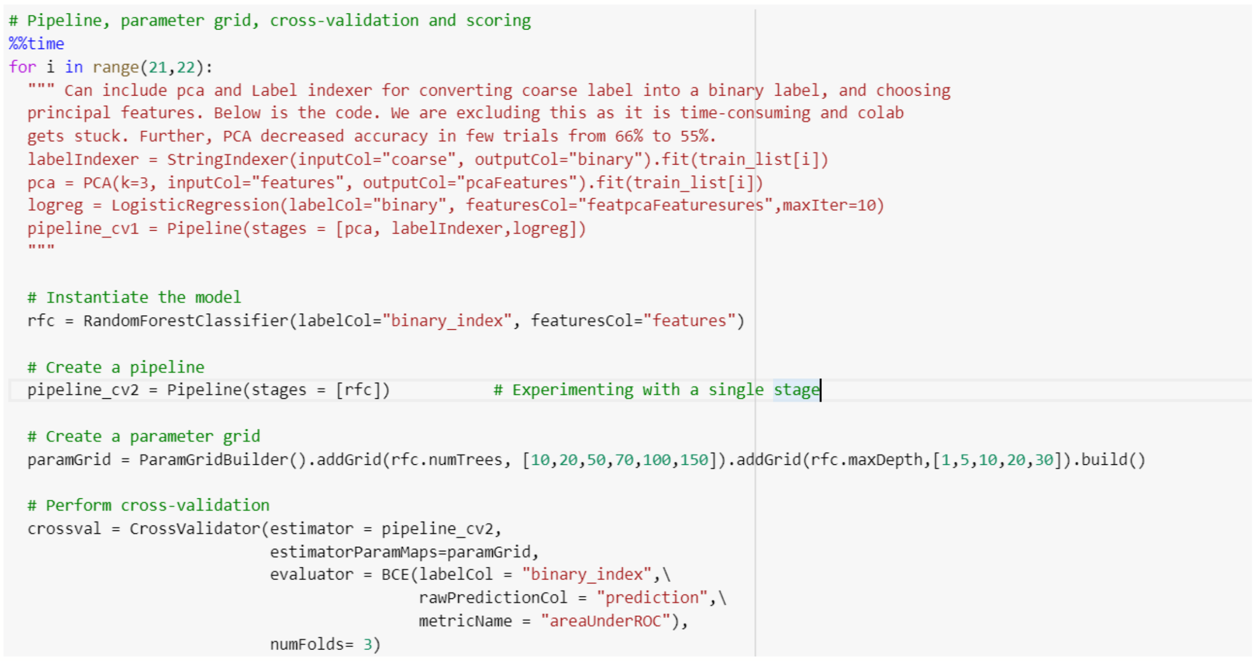
5.2.2.2.1 Hyper-parameter tuning for the best pair

- 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.

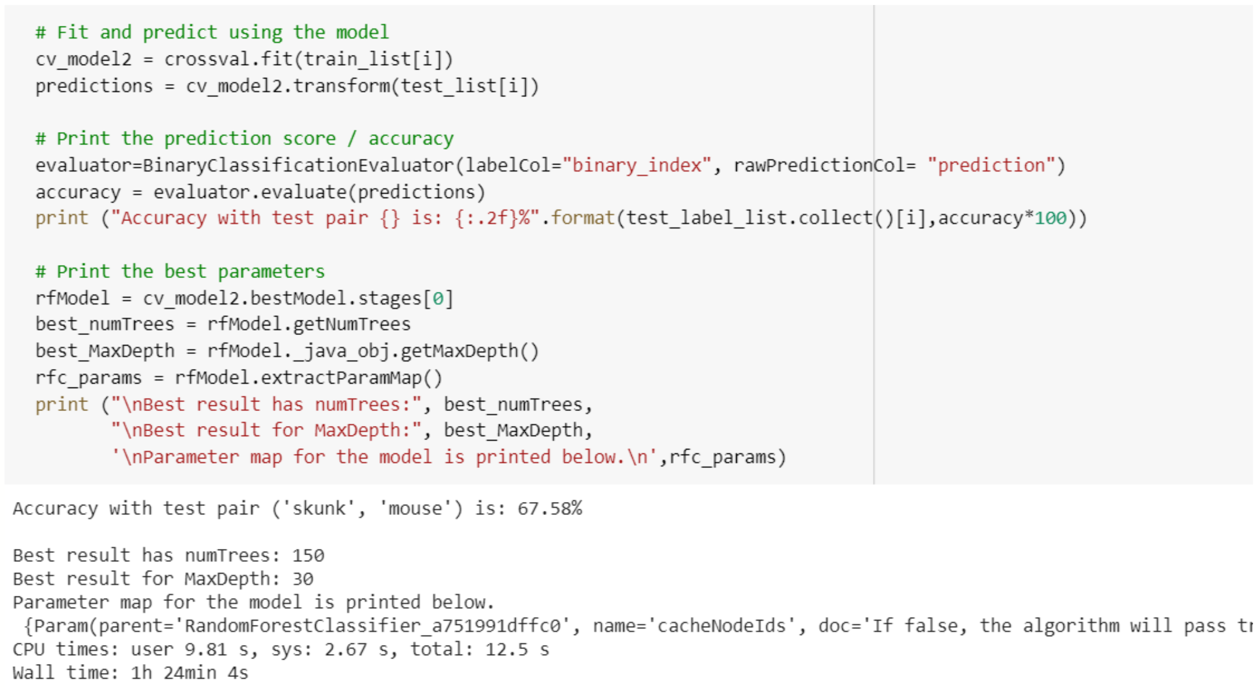
Random Forest:

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



```
# Pipeline, parameter grid, cross-validation and scoring
%%time
for i in range(21,22):
  """ Can include pca and Label indexer for converting coarse label into a binary label, and choosing
  principal features. Below is the code. We are excluding this as it is time-consuming and colab
  gets stuck. Further, PCA decreased accuracy in few trials from 66% to 55%.
  labelIndexer = StringIndexer(inputCol="coarse", outputCol="binary").fit(train_list[i])
  pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures").fit(train_list[i])
  logreg = LogisticRegression(labelCol="binary", featuresCol="featpcaFeaturesures",maxIter=10)
  pipeline_cv1 = Pipeline(stages = [pca, labelIndexer,logreg])
  """

  # Instantiate the model
  rfc = RandomForestClassifier(labelCol="binary_index", featuresCol="features")

  # Create a pipeline
  pipeline_cv2 = Pipeline(stages = [rfc])           # Experimenting with a single stage

  # Create a parameter grid
  paramGrid = ParamGridBuilder().addGrid(rfc.numTrees, [10,20,50,70,100,150]).addGrid(rfc.maxDepth,[1,5,10,20,30]).build()

  # Perform cross-validation
  crossval = CrossValidator(estimator = pipeline_cv2,
                            estimatorParamMaps=paramGrid,
                            evaluator = BCE(labelCol = "binary_index",\
                                            rawPredictionCol = "prediction",\
                                            metricName = "areaUnderROC"),
                            numFolds= 3)
```

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



```python
# Fit and predict using the model
cv_model2 = crossval.fit(train_list[i])
predictions = cv_model2.transform(test_list[i])

# Print the prediction score / accuracy
evaluator=BinaryClassificationEvaluator(labelCol="binary_index", rawPredictionCol= "prediction")
accuracy = evaluator.evaluate(predictions)
print ("Accuracy with test pair {} is: {:.2f}%".format(test_label_list.collect()[i],accuracy*100))

# Print the best parameters
rfModel = cv_model2.bestModel.stages[0]
best_numTrees = rfModel.getNumTrees
best_MaxDepth = rfModel._java_obj.getMaxDepth()
rfc_params = rfModel.extractParamMap()
print ("\nBest result has numTrees:", best_numTrees,
       "\nBest result for MaxDepth:", best_MaxDepth,
       '\nParameter map for the model is printed below.\n',rfc_params)
```

```
Accuracy with test pair ('skunk', 'mouse') is: 67.58%

Best result has numTrees: 150
Best result for MaxDepth: 30
Parameter map for the model is printed below.
 {Param(parent='RandomForestClassifier_a751991dffc0', name='cacheNodeIds', doc='If false, the algorithm will pass t
CPU times: user 9.81 s, sys: 2.67 s, total: 12.5 s
Wall time: 1h 24min 4s
```

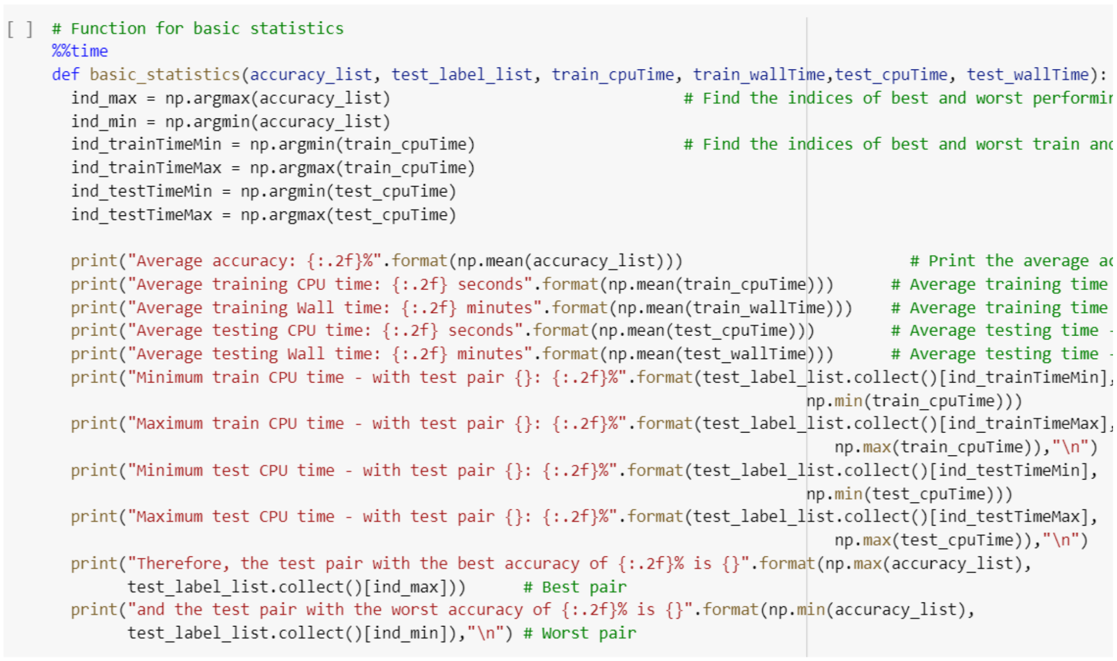
For Random Forest classifier, we tested one pair with 6 values - [10,20,50,70,100,150] for the parameter 'numTrees'. MaxDepth was tested with [1,5,10,20,30]. The model with best accuracy had depth 30 and numTrees 150. We achieved a 2% increase in accuracy. Random forest could not outperform Logistic 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)

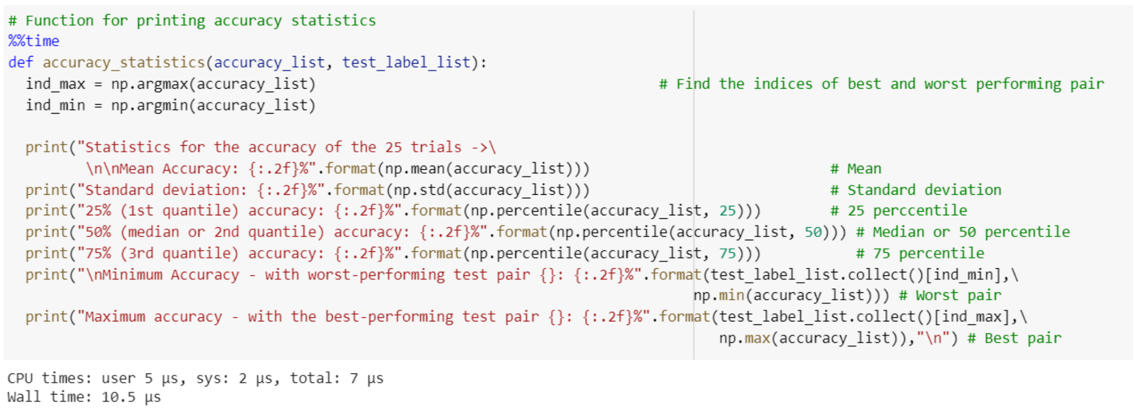
5.3.1 Logistic Regression

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

Function - for printing statistics - To double-check using ***Numpy as well as PySpark***



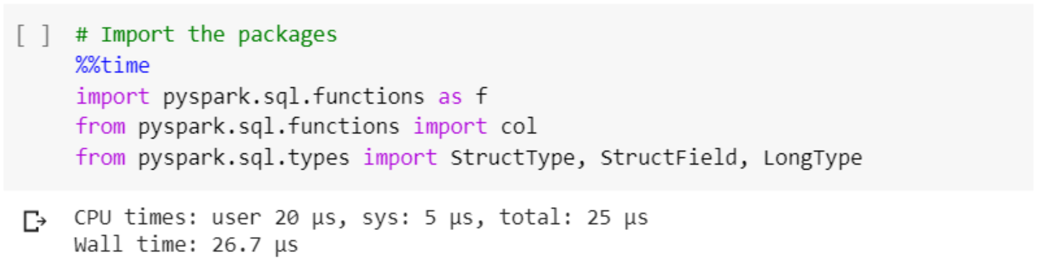
```
[ ]  # Function for basic statistics
     %%time
     def basic_statistics(accuracy_list, test_label_list, train_cpuTime, train_wallTime,test_cpuTime, test_wallTime):
       ind_max = np.argmax(accuracy_list)                               # Find the indices of best and worst performin
       ind_min = np.argmin(accuracy_list)
       ind_trainTimeMin = np.argmin(train_cpuTime)                      # Find the indices of best and worst train and
       ind_trainTimeMax = np.argmax(train_cpuTime)
       ind_testTimeMin = np.argmin(test_cpuTime)
       ind_testTimeMax = np.argmax(test_cpuTime)

       print("Average accuracy: {:.2f}%".format(np.mean(accuracy_list)))                        # Print the average ac
       print("Average training CPU time: {:.2f} seconds".format(np.mean(train_cpuTime)))      # Average training time
       print("Average training Wall time: {:.2f} minutes".format(np.mean(train_wallTime)))    # Average training time
       print("Average testing CPU time: {:.2f} seconds".format(np.mean(test_cpuTime)))        # Average testing time -
       print("Average testing Wall time: {:.2f} minutes".format(np.mean(test_wallTime)))      # Average testing time -
       print("Minimum train CPU time - with test pair {}: {:.2f}%".format(test_label_list.collect()[ind_trainTimeMin],
                                                                          np.min(train_cpuTime)))
       print("Maximum train CPU time - with test pair {}: {:.2f}%".format(test_label_list.collect()[ind_trainTimeMax],
                                                                             np.max(train_cpuTime)),"\n")
       print("Minimum test CPU time - with test pair {}: {:.2f}%".format(test_label_list.collect()[ind_testTimeMin],
                                                                          np.min(test_cpuTime)))
       print("Maximum test CPU time - with test pair {}: {:.2f}%".format(test_label_list.collect()[ind_testTimeMax],
                                                                             np.max(test_cpuTime)),"\n")
       print("Therefore, the test pair with the best accuracy of {:.2f}% is {}".format(np.max(accuracy_list),
             test_label_list.collect()[ind_max]))      # Best pair
       print("and the test pair with the worst accuracy of {:.2f}% is {}".format(np.min(accuracy_list),
             test_label_list.collect()[ind_min]),"\n") # Worst pair
```



```
# Function for printing accuracy statistics
%%time
def accuracy_statistics(accuracy_list, test_label_list):
  ind_max = np.argmax(accuracy_list)                              # Find the indices of best and worst performing pair
  ind_min = np.argmin(accuracy_list)

  print("Statistics for the accuracy of the 25 trials ->\
        \n\nMean Accuracy: {:.2f}%".format(np.mean(accuracy_list)))                      # Mean
  print("Standard deviation: {:.2f}%".format(np.std(accuracy_list)))                     # Standard deviation
  print("25% (1st quantile) accuracy: {:.2f}%".format(np.percentile(accuracy_list, 25)))        # 25 perccentile
  print("50% (median or 2nd quantile) accuracy: {:.2f}%".format(np.percentile(accuracy_list, 50))) # Median or 50 percentile
  print("75% (3rd quantile) accuracy: {:.2f}%".format(np.percentile(accuracy_list, 75)))          # 75 percentile
  print("\nMinimum Accuracy - with worst-performing test pair {}: {:.2f}%".format(test_label_list.collect()[ind_min],\
                                                                   np.min(accuracy_list))) # Worst pair
  print("Maximum accuracy - with the best-performing test pair {}: {:.2f}%".format(test_label_list.collect()[ind_max],\
                                                                     np.max(accuracy_list)),"\n") # Best pair

CPU times: user 5 µs, sys: 2 µs, total: 7 µs
Wall time: 10.5 µs
```

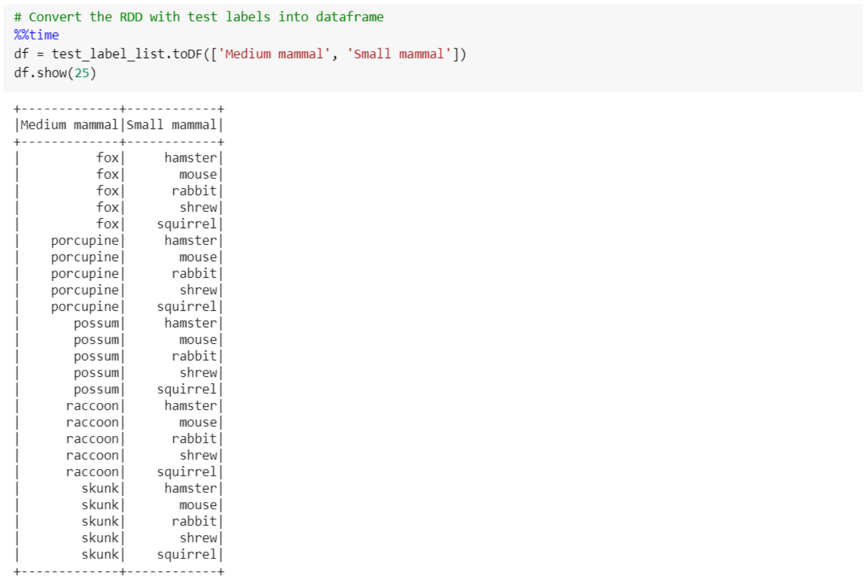
- For Pyspark, we decided to extensively process the results.

Using PySpark



```
[ ]  # Import the packages
     %%time
     import pyspark.sql.functions as f
     from pyspark.sql.functions import col
     from pyspark.sql.types import StructType, StructField, LongType

     CPU times: user 20 µs, sys: 5 µs, total: 25 µs
     Wall time: 26.7 µs
```

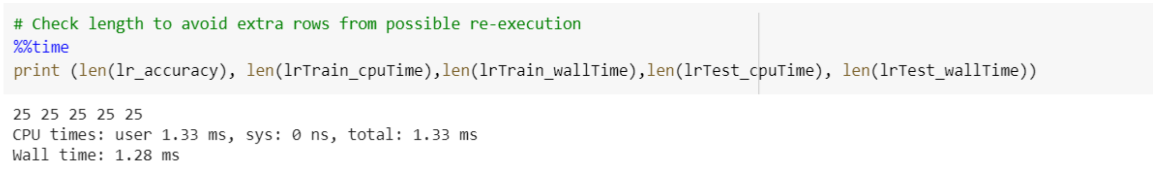
- We converted the RDD with test labels to dataframe.



```
# Convert the RDD with test labels into dataframe
%%time
df = test_label_list.toDF(['Medium mammal', 'Small mammal'])
df.show(25)
```

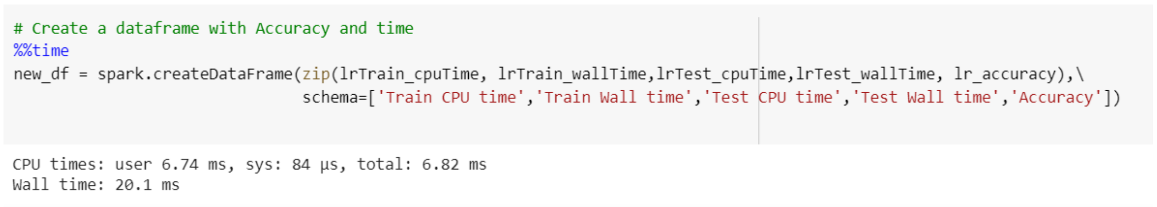
```
+-------------+------------+
|Medium mammal|Small mammal|
+-------------+------------+
|          fox|     hamster|
|          fox|       mouse|
|          fox|      rabbit|
|          fox|       shrew|
|          fox|    squirrel|
|    porcupine|     hamster|
|    porcupine|       mouse|
|    porcupine|      rabbit|
|    porcupine|       shrew|
|    porcupine|    squirrel|
|       possum|     hamster|
|       possum|       mouse|
|       possum|      rabbit|
|       possum|       shrew|
|       possum|    squirrel|
|      raccoon|     hamster|
|      raccoon|       mouse|
|      raccoon|      rabbit|
|      raccoon|       shrew|
|      raccoon|    squirrel|
|        skunk|     hamster|
|        skunk|       mouse|
|        skunk|      rabbit|
|        skunk|       shrew|
|        skunk|    squirrel|
+-------------+------------+
```

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



```
# Check length to avoid extra rows from possible re-execution
%%time
print (len(lr_accuracy), len(lrTrain_cpuTime),len(lrTrain_wallTime),len(lrTest_cpuTime), len(lrTest_wallTime))
```

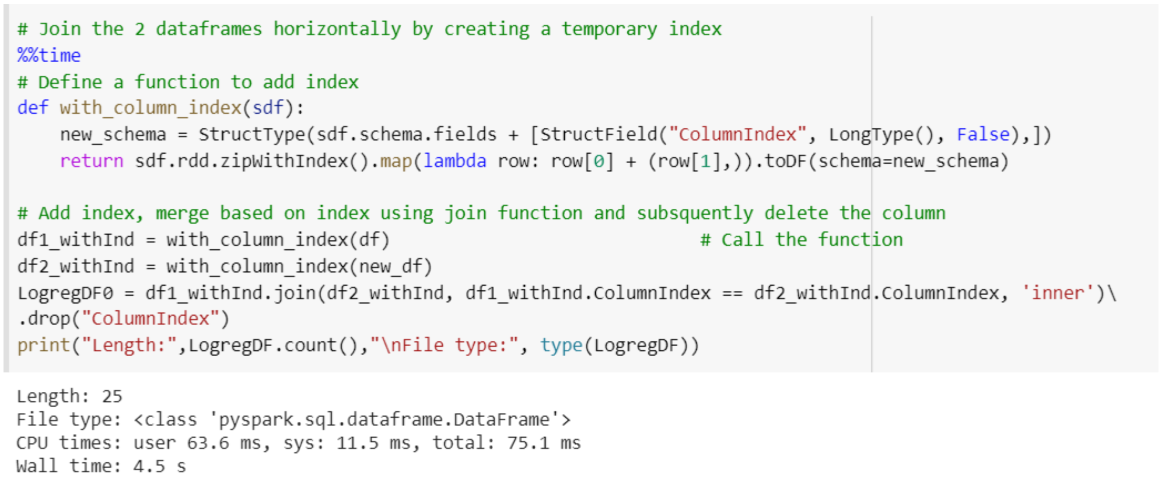
```
25 25 25 25 25
CPU times: user 1.33 ms, sys: 0 ns, total: 1.33 ms
Wall time: 1.28 ms
```



```
# Create a dataframe with Accuracy and time
%%time
new_df = spark.createDataFrame(zip(lrTrain_cpuTime, lrTrain_wallTime,lrTest_cpuTime,lrTest_wallTime, lr_accuracy),\
                               schema=['Train CPU time','Train Wall time','Test CPU time','Test Wall time','Accuracy'])
```

```
CPU times: user 6.74 ms, sys: 84 µs, total: 6.82 ms
Wall time: 20.1 ms
```

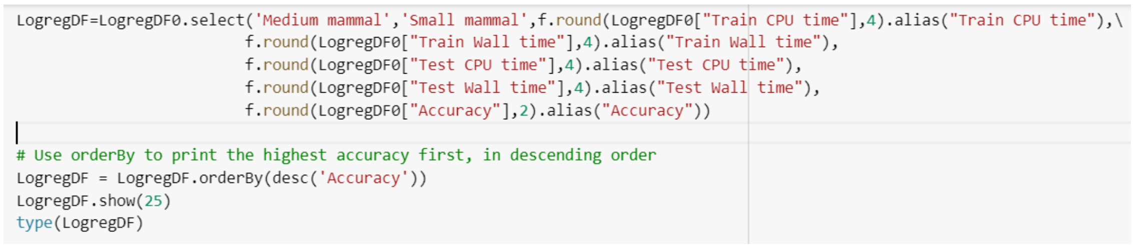
- We joined the tables horizontally using a temporary index number for outer join.



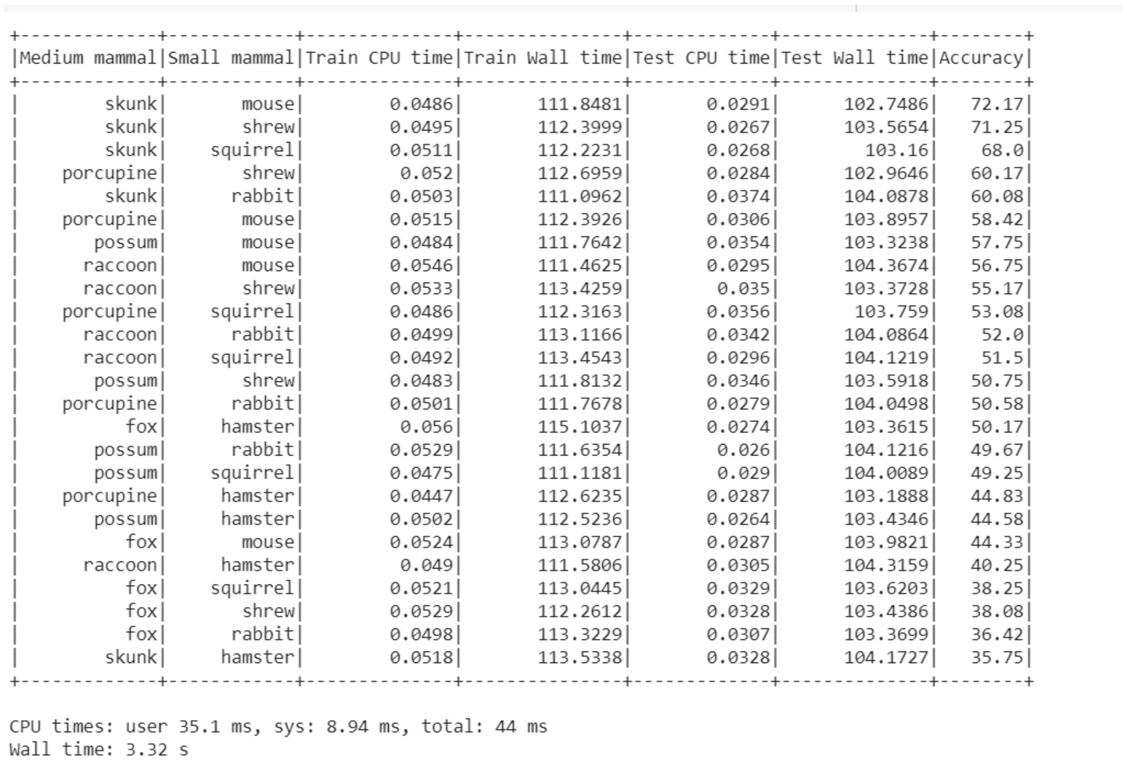
```
# Join the 2 dataframes horizontally by creating a temporary index
%%time
# Define a function to add index
def with_column_index(sdf):
    new_schema = StructType(sdf.schema.fields + [StructField("ColumnIndex", LongType(), False),])
    return sdf.rdd.zipWithIndex().map(lambda row: row[0] + (row[1],)).toDF(schema=new_schema)

# Add index, merge based on index using join function and subsquently delete the column
df1_withInd = with_column_index(df)                                # Call the function
df2_withInd = with_column_index(new_df)
LogregDF0 = df1_withInd.join(df2_withInd, df1_withInd.ColumnIndex == df2_withInd.ColumnIndex, 'inner')\
.drop("ColumnIndex")
print("Length:",LogregDF.count(),"\nFile type:", type(LogregDF))
```

```
Length: 25
File type: <class 'pyspark.sql.dataframe.DataFrame'>
CPU times: user 63.6 ms, sys: 11.5 ms, total: 75.1 ms
Wall time: 4.5 s
```

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

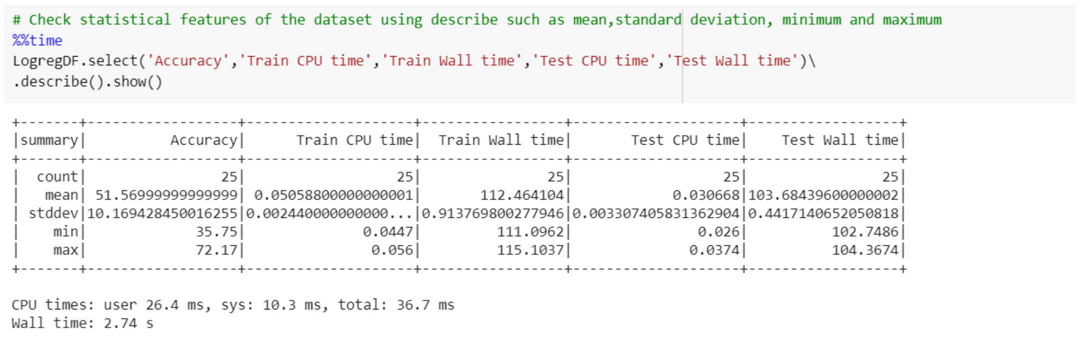


```
LogregDF=LogregDF0.select('Medium mammal','Small mammal',f.round(LogregDF0["Train CPU time"],4).alias("Train CPU time"),\
                         f.round(LogregDF0["Train Wall time"],4).alias("Train Wall time"),
                         f.round(LogregDF0["Test CPU time"],4).alias("Test CPU time"),
                         f.round(LogregDF0["Test Wall time"],4).alias("Test Wall time"),
                         f.round(LogregDF0["Accuracy"],2).alias("Accuracy"))

# Use orderBy to print the highest accuracy first, in descending order
LogregDF = LogregDF.orderBy(desc('Accuracy'))
LogregDF.show(25)
type(LogregDF)
```



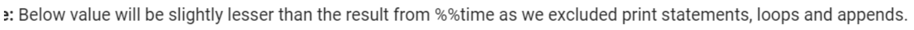
```
+-------------+------------+--------------+---------------+-------------+--------------+--------+
|Medium mammal|Small mammal|Train CPU time|Train Wall time|Test CPU time|Test Wall time|Accuracy|
+-------------+------------+--------------+---------------+-------------+--------------+--------+
|        skunk|       mouse|        0.0486|       111.8481|       0.0291|      102.7486|   72.17|
|        skunk|       shrew|        0.0495|       112.3999|       0.0267|      103.5654|   71.25|
|        skunk|    squirrel|        0.0511|       112.2231|       0.0268|        103.16|    68.0|
|    porcupine|       shrew|         0.052|       112.6959|       0.0284|      102.9646|   60.17|
|        skunk|      rabbit|        0.0503|       111.0962|       0.0374|      104.0878|   60.08|
|    porcupine|       mouse|        0.0515|       112.3926|       0.0306|      103.8957|   58.42|
|       possum|       mouse|        0.0484|       111.7642|       0.0354|      103.3238|   57.75|
|      raccoon|       mouse|        0.0546|       111.4625|       0.0295|      104.3674|   56.75|
|      raccoon|       shrew|        0.0533|       113.4259|        0.035|      103.3728|   55.17|
|    porcupine|    squirrel|        0.0486|       112.3163|       0.0356|       103.759|   53.08|
|      raccoon|      rabbit|        0.0499|       113.1166|       0.0342|      104.0864|    52.0|
|      raccoon|    squirrel|        0.0492|       113.4543|       0.0296|      104.1219|    51.5|
|       possum|       shrew|        0.0483|       111.8132|       0.0346|      103.5918|   50.75|
|    porcupine|      rabbit|        0.0501|       111.7678|       0.0279|      104.0498|   50.58|
|          fox|     hamster|         0.056|       115.1037|       0.0274|      103.3615|   50.17|
|       possum|      rabbit|        0.0529|       111.6354|        0.026|      104.1216|   49.67|
|       possum|    squirrel|        0.0475|       111.1181|        0.029|      104.0089|   49.25|
|    porcupine|     hamster|        0.0447|       112.6235|       0.0287|      103.1888|   44.83|
|       possum|     hamster|        0.0502|       112.5236|       0.0264|      103.4346|   44.58|
|          fox|       mouse|        0.0524|       113.0787|       0.0287|      103.9821|   44.33|
|      raccoon|     hamster|         0.049|       111.5806|       0.0305|      104.3159|   40.25|
|          fox|    squirrel|        0.0521|       113.0445|       0.0329|      103.6203|   38.25|
|          fox|       shrew|        0.0529|       112.2612|       0.0328|      103.4386|   38.08|
|          fox|      rabbit|        0.0498|       113.3229|       0.0307|      103.3699|   36.42|
|        skunk|     hamster|        0.0518|       113.5338|       0.0328|      104.1727|   35.75|
+-------------+------------+--------------+---------------+-------------+--------------+--------+

CPU times: user 35.1 ms, sys: 8.94 ms, total: 44 ms
Wall time: 3.32 s
```

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

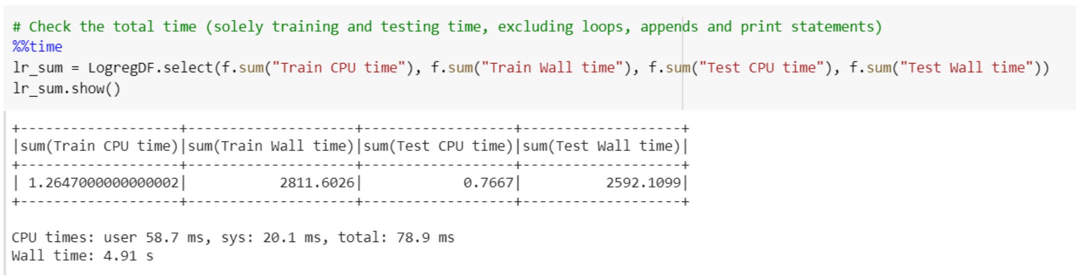


```
# Check statistical features of the dataset using describe such as mean,standard deviation, minimum and maximum
%%time
LogregDF.select('Accuracy','Train CPU time','Train Wall time','Test CPU time','Test Wall time')\
.describe().show()
```

```
+-------+------------------+--------------------+-----------------+--------------------+------------------+
|summary|          Accuracy|      Train CPU time|  Train Wall time|       Test CPU time|    Test Wall time|
+-------+------------------+--------------------+-----------------+--------------------+------------------+
|  count|                25|                  25|               25|                  25|                25|
|   mean| 51.56999999999999| 0.05058800000000001|       112.464104|            0.030668|103.68439600000002|
| stddev|10.169428450016255|0.002440000000000...|0.913769800277946|0.003307405831362904|0.4417140652050818|
|    min|             35.75|              0.0447|         111.0962|               0.026|          102.7486|
|    max|             72.17|               0.056|         115.1037|              0.0374|          104.3674|
+-------+------------------+--------------------+-----------------+--------------------+------------------+

CPU times: user 26.4 ms, sys: 10.3 ms, total: 36.7 ms
Wall time: 2.74 s
```



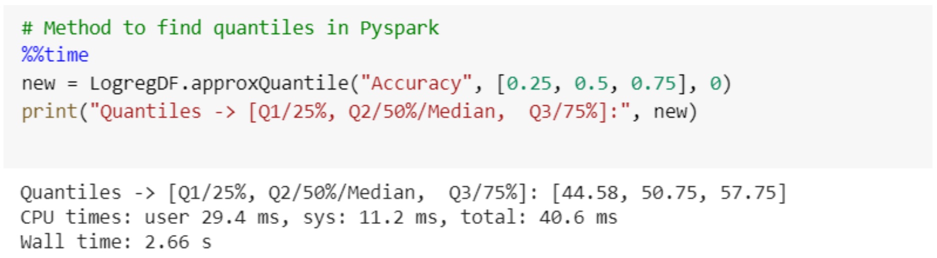
e: Below value will be slightly lesser than the result from %%time as we excluded print statements, loops and appends.



```
# Check the total time (solely training and testing time, excluding loops, appends and print statements)
%%time
lr_sum = LogregDF.select(f.sum("Train CPU time"), f.sum("Train Wall time"), f.sum("Test CPU time"), f.sum("Test Wall time"))
lr_sum.show()
```

```
+-------------------+--------------------+------------------+-------------------+
|sum(Train CPU time)|sum(Train Wall time)|sum(Test CPU time)|sum(Test Wall time)|
+-------------------+--------------------+------------------+-------------------+
| 1.2647000000000002|           2811.6026|            0.7667|          2592.1099|
+-------------------+--------------------+------------------+-------------------+

CPU times: user 58.7 ms, sys: 20.1 ms, total: 78.9 ms
Wall time: 4.91 s
```

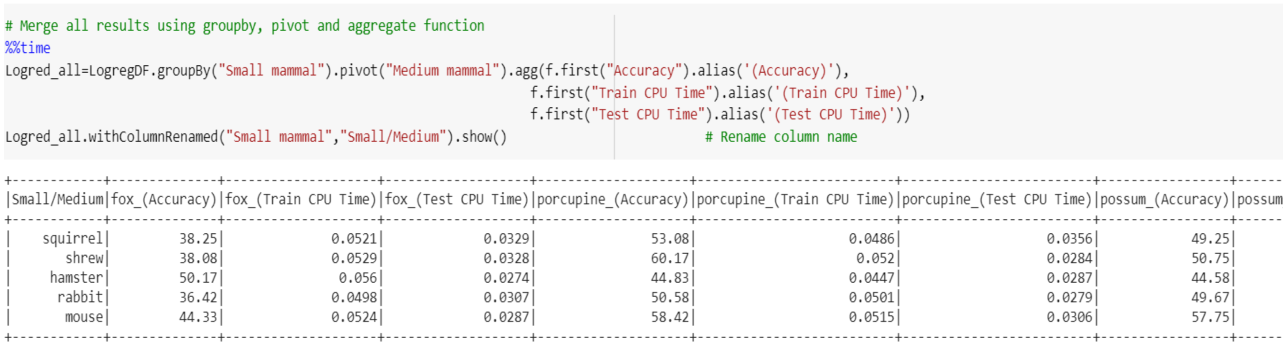
- Quantiles were printed using ***approxquantile()*** function.



```
# Method to find quantiles in Pyspark
%%time
new = LogregDF.approxQuantile("Accuracy", [0.25, 0.5, 0.75], 0)
print("Quantiles -> [Q1/25%, Q2/50%/Median,  Q3/75%]:", new)
```

```
Quantiles -> [Q1/25%, Q2/50%/Median,  Q3/75%]: [44.58, 50.75, 57.75]
CPU times: user 29.4 ms, sys: 11.2 ms, total: 40.6 ms
Wall time: 2.66 s
```

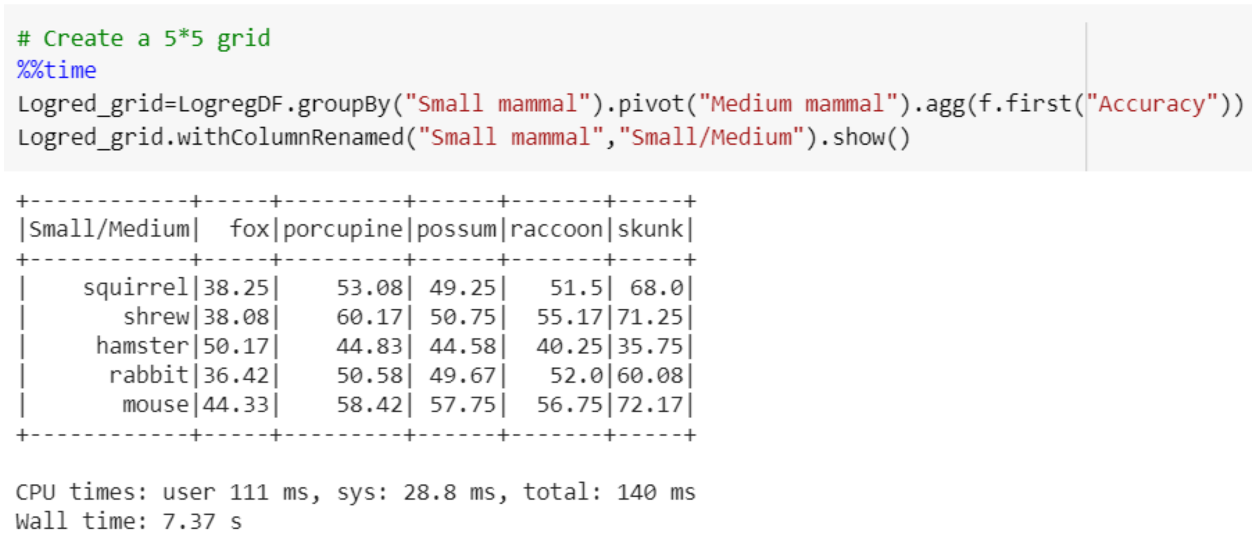
- We merged all results including time and accuracy for each pair, into a 5*15 grid, using ***groupBy, pivot, aggregate, SQL function first*** and ***alias***(for renaming).



```
# Merge all results using groupby, pivot and aggregate function
%%time
Logred_all=LogregDF.groupBy("Small mammal").pivot("Medium mammal").agg(f.first("Accuracy").alias('(Accuracy)'),
                                                                        f.first("Train CPU Time").alias('(Train CPU Time)'),
                                                                        f.first("Test CPU Time").alias('(Test CPU Time)'))
Logred_all.withColumnRenamed("Small mammal","Small/Medium").show()                       # Rename column name
```

```
+------------+-------------+-------------------+------------------+------------------+------------------------+-----------------------+----------------+------
|Small/Medium|fox_(Accuracy)|fox_(Train CPU Time)|fox_(Test CPU Time)|porcupine_(Accuracy)|porcupine_(Train CPU Time)|porcupine_(Test CPU Time)|possum_(Accuracy)|possum
+------------+-------------+-------------------+------------------+------------------+------------------------+-----------------------+----------------+------
|    squirrel|        38.25|             0.0521|            0.0329|             53.08|                  0.0486|                 0.0356|           49.25|
|       shrew|        38.08|             0.0529|            0.0328|             60.17|                   0.052|                 0.0284|           50.75|
|     hamster|        50.17|              0.056|            0.0274|             44.83|                  0.0447|                 0.0287|           44.58|
|      rabbit|        36.42|             0.0498|            0.0307|             50.58|                  0.0501|                 0.0279|           49.67|
|       mouse|        44.33|             0.0524|            0.0287|             58.42|                  0.0515|                 0.0306|           57.75|
+------------+-------------+-------------------+------------------+------------------+------------------------+-----------------------+----------------+------
```

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



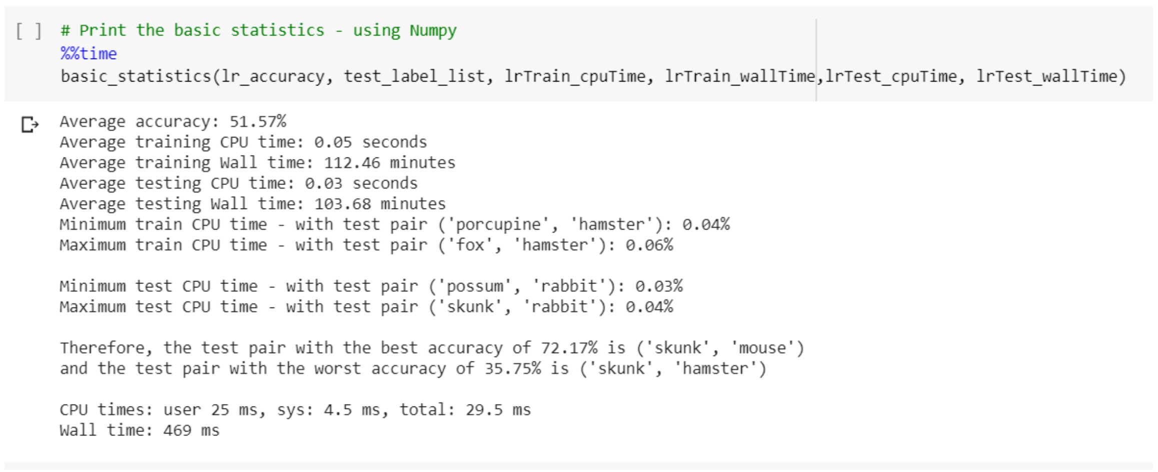
```
# Create a 5*5 grid
%%time
Logred_grid=LogregDF.groupBy("Small mammal").pivot("Medium mammal").agg(f.first("Accuracy"))
Logred_grid.withColumnRenamed("Small mammal","Small/Medium").show()
```

```
+------------+-----+---------+------+-------+-----+
|Small/Medium|  fox|porcupine|possum|raccoon|skunk|
+------------+-----+---------+------+-------+-----+
|    squirrel|38.25|    53.08| 49.25|   51.5| 68.0|
|       shrew|38.08|    60.17| 50.75|  55.17|71.25|
|     hamster|50.17|    44.83| 44.58|  40.25|35.75|
|      rabbit|36.42|    50.58| 49.67|   52.0|60.08|
|       mouse|44.33|    58.42| 57.75|  56.75|72.17|
+------------+-----+---------+------+-------+-----+

CPU times: user 111 ms, sys: 28.8 ms, total: 140 ms
Wall time: 7.37 s
```

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

Model 1: Statistical study - we are double-checking using ***Numpy as well as PySpark***

Using Numpy

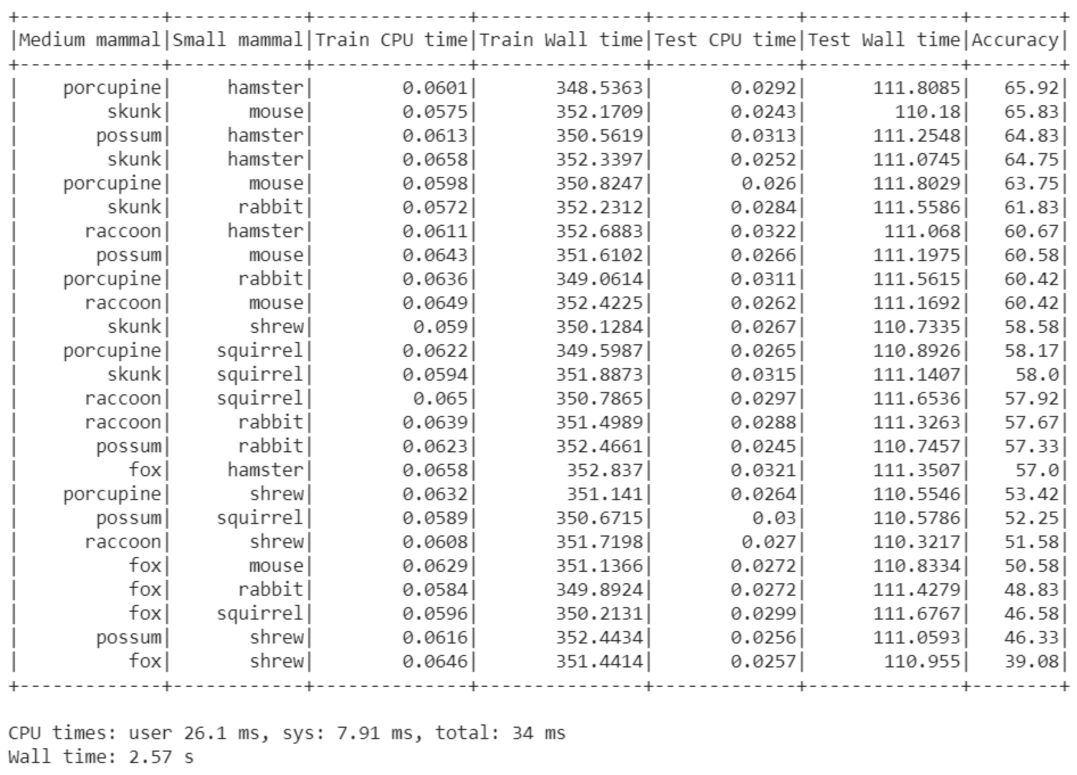


```
[ ] # Print the basic statistics - using Numpy
    %%time
    basic_statistics(lr_accuracy, test_label_list, lrTrain_cpuTime, lrTrain_wallTime,lrTest_cpuTime, lrTest_wallTime)
```

```
Average accuracy: 51.57%
Average training CPU time: 0.05 seconds
Average training Wall time: 112.46 minutes
Average testing CPU time: 0.03 seconds
Average testing Wall time: 103.68 minutes
Minimum train CPU time - with test pair ('porcupine', 'hamster'): 0.04%
Maximum train CPU time - with test pair ('fox', 'hamster'): 0.06%

Minimum test CPU time - with test pair ('possum', 'rabbit'): 0.03%
Maximum test CPU time - with test pair ('skunk', 'rabbit'): 0.04%

Therefore, the test pair with the best accuracy of 72.17% is ('skunk', 'mouse')
and the test pair with the worst accuracy of 35.75% is ('skunk', 'hamster')

CPU times: user 25 ms, sys: 4.5 ms, total: 29.5 ms
Wall time: 469 ms
```

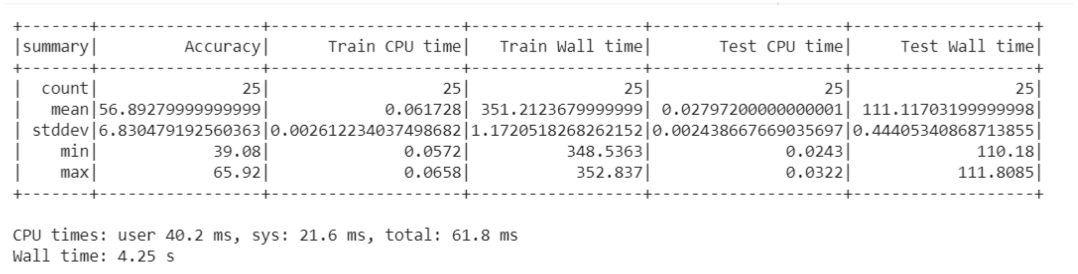
5.3.2 Random Forest

- Similar manipulations were performed to derive the below results.

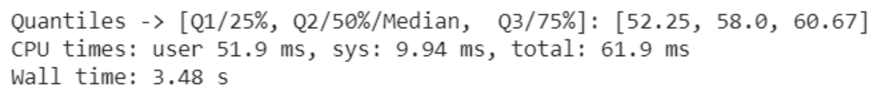


```
+-------------+-----------+--------------+---------------+-------------+--------------+--------+
|Medium mammal|Small mammal|Train CPU time|Train Wall time|Test CPU time|Test Wall time|Accuracy|
+-------------+-----------+--------------+---------------+-------------+--------------+--------+
|    porcupine|    hamster|        0.0601|       348.5363|       0.0292|      111.8085|   65.92|
|        skunk|      mouse|        0.0575|       352.1709|       0.0243|        110.18|   65.83|
|       possum|    hamster|        0.0613|       350.5619|       0.0313|      111.2548|   64.83|
|        skunk|    hamster|        0.0658|       352.3397|       0.0252|      111.0745|   64.75|
|    porcupine|      mouse|        0.0598|       350.8247|        0.026|      111.8029|   63.75|
|        skunk|     rabbit|        0.0572|       352.2312|       0.0284|      111.5586|   61.83|
|      raccoon|    hamster|        0.0611|       352.6883|       0.0322|       111.068|   60.67|
|       possum|      mouse|        0.0643|       351.6102|       0.0266|      111.1975|   60.58|
|    porcupine|     rabbit|        0.0636|       349.0614|       0.0311|      111.5615|   60.42|
|      raccoon|      mouse|        0.0649|       352.4225|       0.0262|      111.1692|   60.42|
|        skunk|      shrew|         0.059|       350.1284|       0.0267|      110.7335|   58.58|
|    porcupine|   squirrel|        0.0622|       349.5987|       0.0265|      110.8926|   58.17|
|        skunk|   squirrel|        0.0594|       351.8873|       0.0315|      111.1407|    58.0|
|      raccoon|   squirrel|         0.065|       350.7865|       0.0297|      111.6536|   57.92|
|      raccoon|     rabbit|        0.0639|       351.4989|       0.0288|      111.3263|   57.67|
|       possum|     rabbit|        0.0623|       352.4661|       0.0245|      110.7457|   57.33|
|          fox|    hamster|        0.0658|        352.837|       0.0321|      111.3507|    57.0|
|    porcupine|      shrew|        0.0632|        351.141|       0.0264|      110.5546|   53.42|
|       possum|   squirrel|        0.0589|       350.6715|         0.03|      110.5786|   52.25|
|      raccoon|      shrew|        0.0608|       351.7198|        0.027|      110.3217|   51.58|
|          fox|      mouse|        0.0629|       351.1366|       0.0272|      110.8334|   50.58|
|          fox|     rabbit|        0.0584|       349.8924|       0.0272|      111.4279|   48.83|
|          fox|   squirrel|        0.0596|       350.2131|       0.0299|      111.6767|   46.58|
|       possum|      shrew|        0.0616|       352.4434|       0.0256|      111.0593|   46.33|
|          fox|      shrew|        0.0646|       351.4414|       0.0257|       110.955|   39.08|
+-------------+-----------+--------------+---------------+-------------+--------------+--------+

CPU times: user 26.1 ms, sys: 7.91 ms, total: 34 ms
Wall time: 2.57 s
```

- Statistical features and quantiles are as follows.

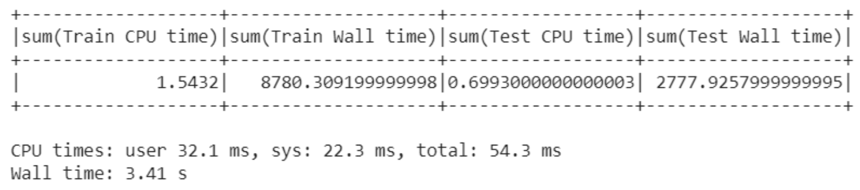


```
+-------+-----------------+-------------------+-----------------+--------------------+-------------------+
|summary|         Accuracy|     Train CPU time|  Train Wall time|       Test CPU time|     Test Wall time|
+-------+-----------------+-------------------+-----------------+--------------------+-------------------+
|  count|               25|                 25|               25|                  25|                 25|
|   mean|56.89279999999999|           0.061728| 351.2123679999999| 0.02797200000000001| 111.11703199999998|
| stddev|6.830479192560363|0.002612234037498682|1.1720518268262152|0.002438667669035697|0.44405340868713855|
|    min|            39.08|             0.0572|         348.5363|              0.0243|             110.18|
|    max|            65.92|             0.0658|          352.837|              0.0322|           111.8085|
+-------+-----------------+-------------------+-----------------+--------------------+-------------------+

CPU times: user 40.2 ms, sys: 21.6 ms, total: 61.8 ms
Wall time: 4.25 s
```



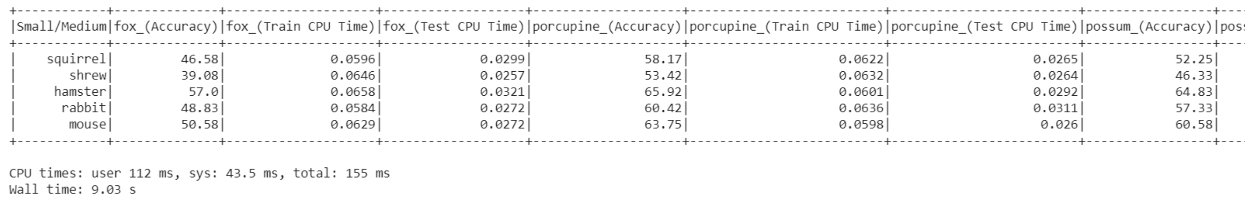
```
Quantiles -> [Q1/25%, Q2/50%/Median,  Q3/75%]: [52.25, 58.0, 60.67]
CPU times: user 51.9 ms, sys: 9.94 ms, total: 61.9 ms
Wall time: 3.48 s
```

- Sum of CPU and wall times are as follows.



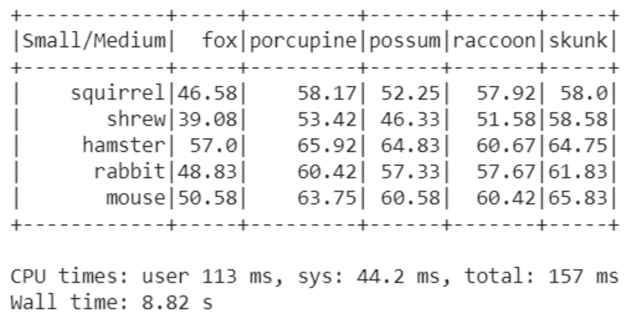
```
+------------------+-------------------+------------------+-------------------+
|sum(Train CPU time)|sum(Train Wall time)|sum(Test CPU time)|sum(Test Wall time)|
+------------------+-------------------+------------------+-------------------+
|            1.5432|   8780.309199999998|0.6993000000000003| 2777.9257999999995|
+------------------+-------------------+------------------+-------------------+

CPU times: user 32.1 ms, sys: 22.3 ms, total: 54.3 ms
Wall time: 3.41 s
```

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



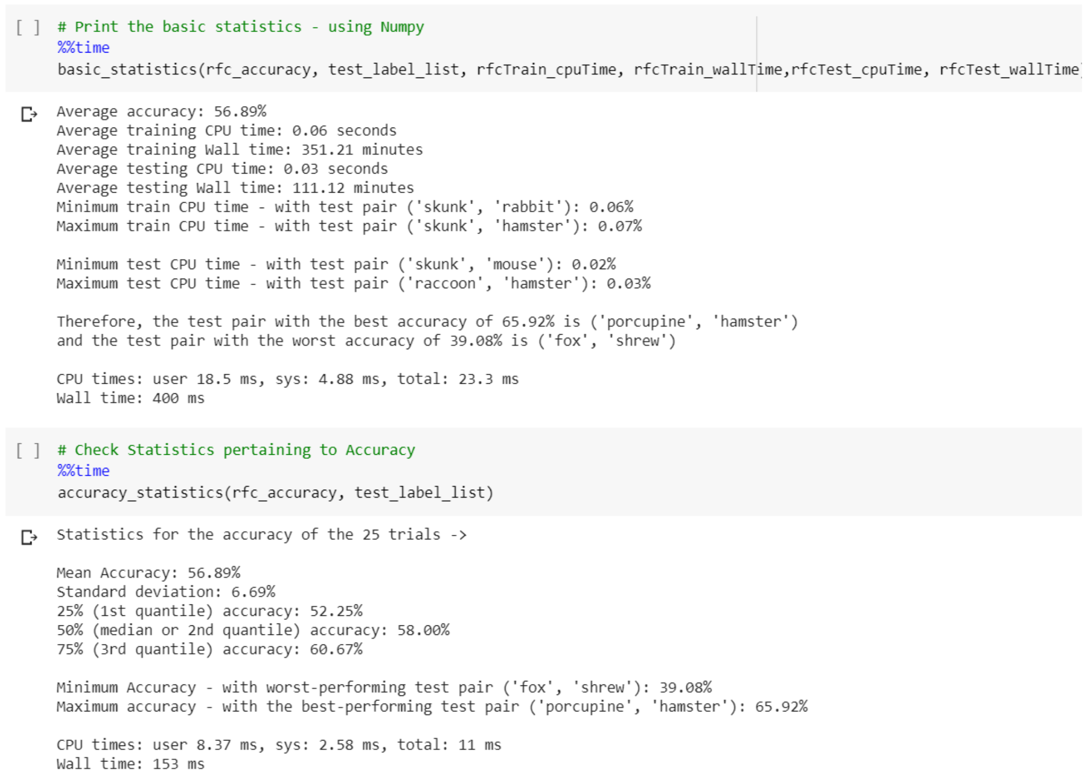
```
+------------+-------------+-------------------+------------------+-------------------+-------------------------+------------------------+----------------+---
|Small/Medium|fox_(Accuracy)|fox_(Train CPU Time)|fox_(Test CPU Time)|porcupine_(Accuracy)|porcupine_(Train CPU Time)|porcupine_(Test CPU Time)|possum_(Accuracy)|pos
+------------+-------------+-------------------+------------------+-------------------+-------------------------+------------------------+----------------+---
|    squirrel|        46.58|             0.0596|            0.0299|              58.17|                   0.0622|                  0.0265|           52.25|
|       shrew|        39.08|             0.0646|            0.0257|              53.42|                   0.0632|                  0.0264|           46.33|
|     hamster|         57.0|             0.0658|            0.0321|              65.92|                   0.0601|                  0.0292|           64.83|
|      rabbit|        48.83|             0.0584|            0.0272|              60.42|                   0.0636|                  0.0311|           57.33|
|       mouse|        50.58|             0.0629|            0.0272|              63.75|                   0.0598|                   0.026|           60.58|
+------------+-------------+-------------------+------------------+-------------------+-------------------------+------------------------+----------------+---

CPU times: user 112 ms, sys: 43.5 ms, total: 155 ms
Wall time: 9.03 s
```

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



```
+------------+-----+---------+------+-------+-----+
|Small/Medium|  fox|porcupine|possum|raccoon|skunk|
+------------+-----+---------+------+-------+-----+
|    squirrel|46.58|    58.17| 52.25|  57.92| 58.0|
|       shrew|39.08|    53.42| 46.33|  51.58|58.58|
|     hamster| 57.0|    65.92| 64.83|  60.67|64.75|
|      rabbit|48.83|    60.42| 57.33|  57.67|61.83|
|       mouse|50.58|    63.75| 60.58|  60.42|65.83|
+------------+-----+---------+------+-------+-----+

CPU times: user 113 ms, sys: 44.2 ms, total: 157 ms
Wall time: 8.82 s
```

- Statistics generated by Numpy were exactly the same.



```
[ ] # Print the basic statistics - using Numpy
    %%time
    basic_statistics(rfc_accuracy, test_label_list, rfcTrain_cpuTime, rfcTrain_wallTime,rfcTest_cpuTime, rfcTest_wallTime
```

```
Average accuracy: 56.89%
Average training CPU time: 0.06 seconds
Average training Wall time: 351.21 minutes
Average testing CPU time: 0.03 seconds
Average testing Wall time: 111.12 minutes
Minimum train CPU time - with test pair ('skunk', 'rabbit'): 0.06%
Maximum train CPU time - with test pair ('skunk', 'hamster'): 0.07%

Minimum test CPU time - with test pair ('skunk', 'mouse'): 0.02%
Maximum test CPU time - with test pair ('raccoon', 'hamster'): 0.03%

Therefore, the test pair with the best accuracy of 65.92% is ('porcupine', 'hamster')
and the test pair with the worst accuracy of 39.08% is ('fox', 'shrew')

CPU times: user 18.5 ms, sys: 4.88 ms, total: 23.3 ms
Wall time: 400 ms
```

```
[ ] # Check Statistics pertaining to Accuracy
    %%time
    accuracy_statistics(rfc_accuracy, test_label_list)
```

```
Statistics for the accuracy of the 25 trials ->

Mean Accuracy: 56.89%
Standard deviation: 6.69%
25% (1st quantile) accuracy: 52.25%
50% (median or 2nd quantile) accuracy: 58.00%
75% (3rd quantile) accuracy: 60.67%

Minimum Accuracy - with worst-performing test pair ('fox', 'shrew'): 39.08%
Maximum accuracy - with the best-performing test pair ('porcupine', 'hamster'): 65.92%

CPU times: user 8.37 ms, sys: 2.58 ms, total: 11 ms
Wall time: 153 ms
```

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.

6. 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	matplotlib(same)

		plt.imshow(img) Define grid function add the fine label and coarse label	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
			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()

		coarse label	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 s core / accuracy	print ("Logistic Regressio n Accuracy: {}%".for mat(lr.sco re(x_test_ 1, y_test_bin)*100))	BinaryClassificationEvaluator' or multiclassclassificationevaluator can be used for finding the accuracy of the 3 models.
			
			eval=BinaryClassificationEvaluator(l abelCol="binary_index", rawPredicti onCol= "prediction")
			
			accuracy = (eval.evaluate(predict_lr)) *100
			print("Model Accuracy: %.3f%%" % accuracy)

Model 2 – Naïve Bayes Classifier	Import package	from sklearn.na ive_bayes import Multinom ialNB	from pyspark.ml.classification import NaiveBayes
	Instantiate the model	naive = Multinom ialNB()	naive_bayes = NaiveBayes(featuresC ol="features", labelCol="binary_inde x",smoothing=1.0, modelType="multi nomial")
	Fit the model	naive.fit(x _train_1, y_train_bi n)	naive_bayes = naive_bayes.fit(train_d f)

	Predict using the model and print the first 10 rows from the resultant dataframe.	naive_pre dict= naive.pred ict(x_test_ 1)
		
		cifar_grid (x_test_1, y_test_bin
			predict_nb = naive_bayes.transform(t est_df)
			
			predict_nb.select("coarse","binary_in dex","prediction","probability").show (10)
			
			Total execution time for this model was more than 3 minutes.

		,indices,4, naive_predict)	
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,	**Method 1** -Simple version using below code by converting the dataframe to RDD and using zipWithIndex and countByKey functions.

		sample_weight=None)	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

7. 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:

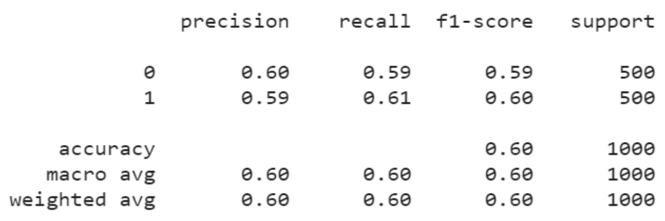
Logistic Regression Confusion Matrix Comparison

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

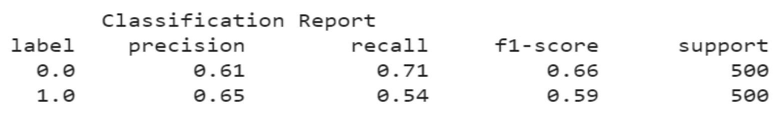
Table 5.a.1 Table of Logistic Regression Confusion Matrix Comparison

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

Scikit-Learn **PySpark**



```
              precision    recall  f1-score   support

           0       0.60      0.59      0.59       500
           1       0.59      0.61      0.60       500

    accuracy                           0.60      1000
   macro avg       0.60      0.60      0.60      1000
weighted avg       0.60      0.60      0.60      1000
```



```
         Classification Report
label    precision         recall     f1-score       support
  0.0         0.61           0.71         0.66           500
  1.0         0.65           0.54         0.59           500
```

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

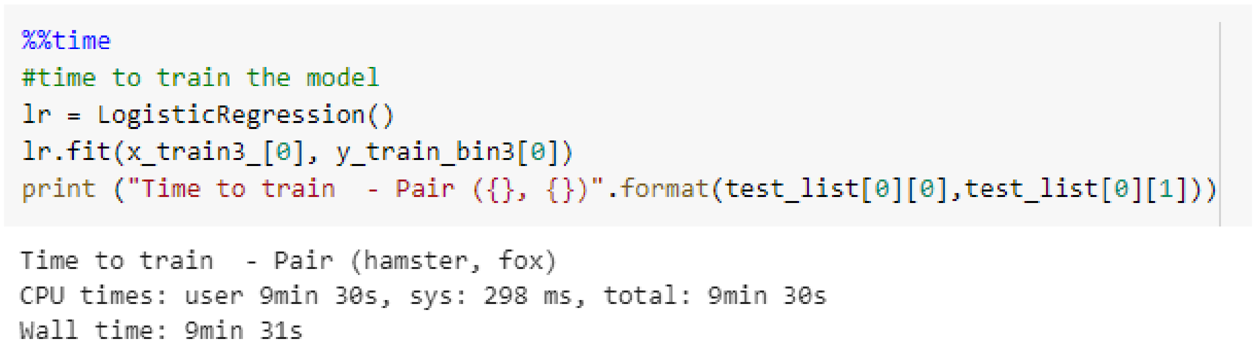
7.1.2 Milestone 2 - Prediction on one testing subclass images from each of the two superclasses Results

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.

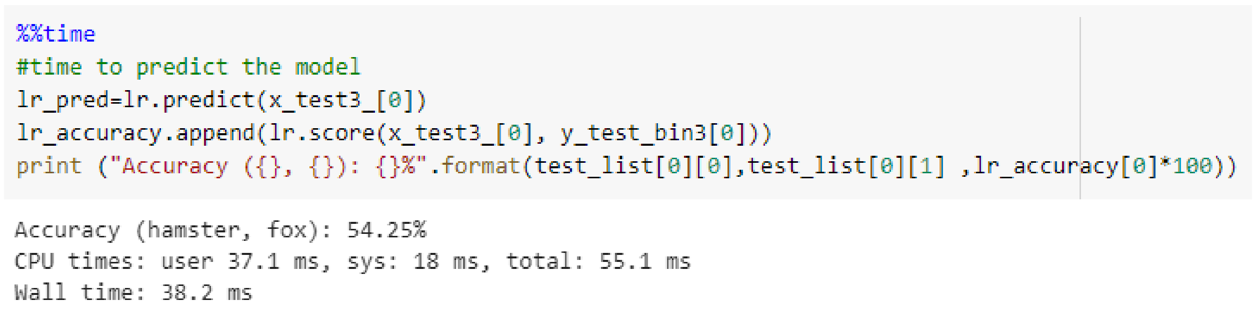
Through this exercise it was detected, as expected, that most of the time goes to training the model with the prediction happening in mili-seconds.

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



```
%%time
#time to train the model
lr = LogisticRegression()
lr.fit(x_train3_[0], y_train_bin3[0])
print ("Time to train  - Pair ({}, {})".format(test_list[0][0],test_list[0][1]))
```

```
Time to train  - Pair (hamster, fox)
CPU times: user 9min 30s, sys: 298 ms, total: 9min 30s
Wall time: 9min 31s
```



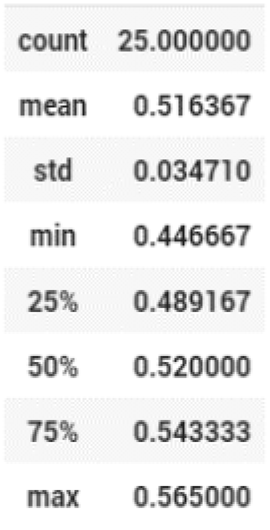
```
%%time
#time to predict the model
lr_pred=lr.predict(x_test3_[0])
lr_accuracy.append(lr.score(x_test3_[0], y_test_bin3[0]))
print ("Accuracy ({}, {}): {}%".format(test_list[0][0],test_list[0][1] ,lr_accuracy[0]*100))
```

```
Accuracy (hamster, fox): 54.25%
CPU times: user 37.1 ms, sys: 18 ms, total: 55.1 ms
Wall time: 38.2 ms
```

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.



count	25.000000
mean	0.516367
std	0.034710
min	0.446667
25%	0.489167
50%	0.520000
75%	0.543333
max	0.565000

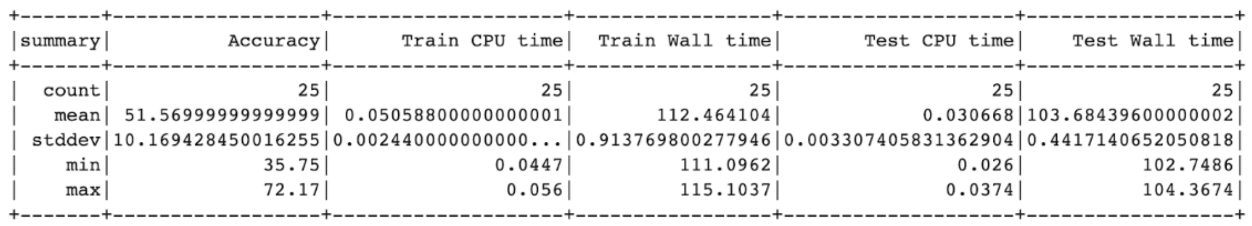
Regarding the training time, on average it took 434s, with a standard deviation of 222s. The longest took 966s and the fastest 175s.

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:



```
+-------+------------------+--------------------+-----------------+--------------------+------------------+
|summary|          Accuracy|      Train CPU time|  Train Wall time|       Test CPU time|    Test Wall time|
+-------+------------------+--------------------+-----------------+--------------------+------------------+
|  count|                25|                  25|               25|                  25|                25|
|   mean| 51.56999999999999| 0.05058800000000001|       112.464104|            0.030668|103.68439600000002|
| stddev|10.169428450016255|0.002440000000000...|0.913769800277946|0.003307405831362904|0.4417140652050818|
|    min|             35.75|              0.0447|         111.0962|               0.026|          102.7486|
|    max|             72.17|               0.056|         115.1037|              0.0374|          104.3674|
+-------+------------------+--------------------+-----------------+--------------------+------------------+
```

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

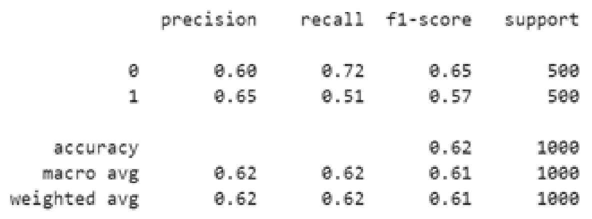
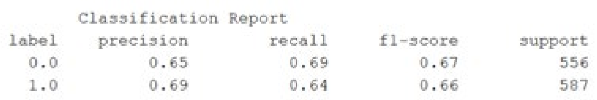
The comparison of Confusion Matrix and Classification Report for Scikit-Learn vs. Spark for our Milestone 1 results are shown below:

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**
	

```
              precision    recall  f1-score   support

           0       0.60      0.72      0.65       500
           1       0.65      0.51      0.57       500

    accuracy                           0.62      1000
   macro avg       0.62      0.62      0.61      1000
weighted avg       0.62      0.62      0.61      1000
```

```
          Classification Report
label    precision          recall      f1-score         support
 0.0        0.65             0.69         0.67             556
 1.0        0.69             0.64         0.66             587
```

● 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.

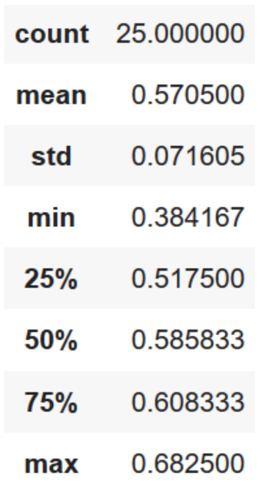
7.2.2 Milestone 2 - Prediction on One Testing Subclass Images From Each of the Two Superclasses Results

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:



count	25.000000
mean	0.570500
std	0.071605
min	0.384167
25%	0.517500
50%	0.585833
75%	0.608333
max	0.682500

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:

```
+-------+-----------------+--------------------+------------------+--------------------+-------------------+
|summary|         Accuracy|      Train CPU time|   Train Wall time|        Test CPU time|     Test Wall time|
+-------+-----------------+--------------------+------------------+--------------------+-------------------+
|  count|               25|                  25|                25|                  25|                 25|
|   mean|56.89279999999999|            0.061728| 351.2123679999999| 0.02797200000000001| 111.11703199999998|
| stddev|6.830479192560363|0.002612234037498682|1.1720518268262152|0.002438667669035697|0.44405340868713855|
|    min|            39.08|              0.0572|          348.5363|              0.0243|             110.18|
|    max|            65.92|              0.0658|           352.837|              0.0322|           111.8085|
+-------+-----------------+--------------------+------------------+--------------------+-------------------+
```

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 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

8. 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:

a. 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

b. 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