Flight Delays Prediction

Khoa Le Nguyen, Renny Octavia Tan
University of Stavanger, Stavanger, Rogaland 4036, Norway,
k.lenguyen@stud.uis.no, renny.octavia@gmail.com

April 22, 2019

Abstract

Delayed or cancelled flights can cost travelers in the U.S. more than $25 billion a year[1]. That leads to the need of an accurate prediction model for flight delays. In this project, we use Random Forest as the main algorithm to build the model. Due to the big amount of flight data, we implement the project using MRJob, PySpark and Spark's MLlib then compare the performance and accuracy of those implementations.

`Git repository: https://bitbucket.org/dataintensivesystem/fdp/src/master/`

1 Use Case and Data set

Every year, there's more than 20% of airline flights are delayed or cancelled (figure 1)[2] and that costs lots of money. This inspires us to build a predicting model to predict whether flights will be delayed or not and for how long will be the delay. We formulate this as a regression problem. We use the dataset from The Bureau of Transportation Statistics[3]. They have a deluge on-time arrival and departure data for non-stop domestic flights in U.S. by month and year, by carrier and by origin and destination airport. Includes scheduled and actual departure and arrival times, cancelled and diverted flights, taxi-out and taxi-in times, causes of delay and cancellation, air time, and non-stop distance.

Since not all data fields are relevant for the predicting model, we decide just to download the data with 29 fields. We download 3.6GB data of the last 4 years from 2015 to 2018 which consists of more than 23 million flight records. Since they divide the data into months, we have to concatenate multiple month data files into one training file (consists of flight records in 3 years 2015-2017) and one test file (flight data in 2018). Our programs can predict delay time (in minutes) on both arrival and departure time.

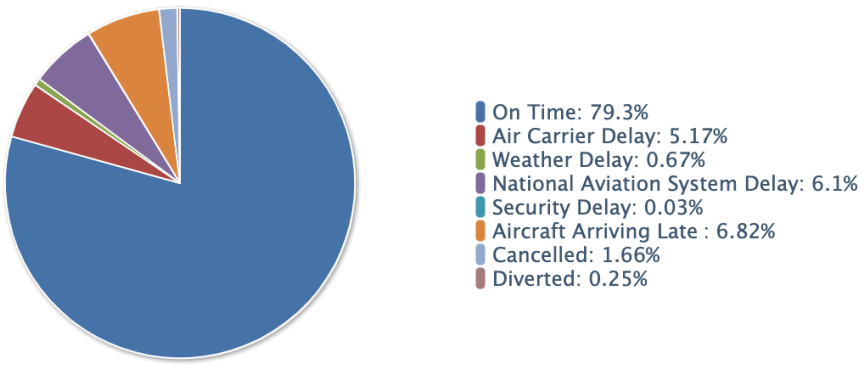

Figure 1: On-Time Arrival Performance National (January - November, 2018)

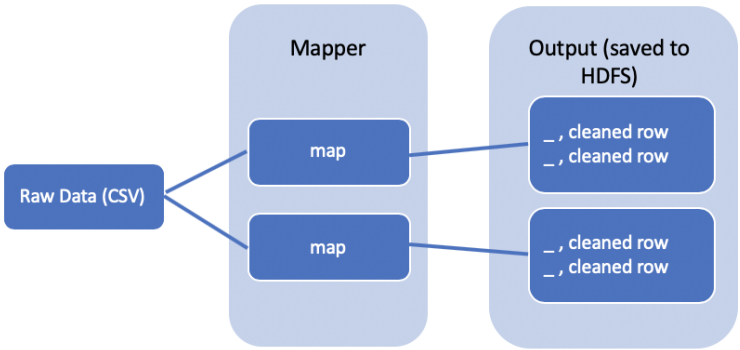

Figure 2: Pre-processing process MRJob-Hadoop

2 Pre-processing

As the dataset for our case has good structure and we can choose which features we want when downloading the data, the pre-processing part has become quite straightforward. In MRJob-Hadoop, the pre-processing is done with one mapper-only job as seen in fig 2.

```
class MRPreprocessing(MRJob):
    def mapper(self,_,line):
        line = line.replace('"', '').split(',')
        line = map(str.strip, line)
        if len(line) >= 29 and line[0] != 'YEAR' and line[16] != '' and line[11] !='':
            for i in [2,9,14,20,23]:
                try:
                    isnumber = float(line[i])
                except ValueError:
                    if i==9 or i==14:
                        lines[i] = '0000'
                    else:
                        lines[i]='0.00'

            crs_dep_time = int(line[9][0:2])*60 + int(line[9][2:])
            crs_arr_time = int(line[14][0:2])*60 + int(line[14][2:])
            data_row = [line[1],line[2],line[3],line[4],line[5],line[6],line[7],
            line[8],crs_dep_time,crs_arr_time,line[20],line[23],line[11],line[16]]

            yield _, ','.join(map(str, data_row))
```

Listing 1: Pre-processing MRJob code

The implementation of pre-processing part is similar in hadoop and spark. The task consists of:

- **Clean the data-** Remove any unwanted or unused inform characters, white-spaces, brackets.
- **Handling missing values-** Exclude the line with empty arrival delay and departure delay, which indicates that the flight was cancelled. Note that the arrival delay and departure is in form of minutes, where departure and arrival ahead of time is also recorded in minutes with minus sign in front of it.
- **Convert data-** convert the column values to the format that will be useful for the machine learning algorithm. Such as: converting departure time and arrival time column which have format of hhmm to number of minutes relative to 00:00 (h*60 + m).

```
1  def convert(lines):
2      for i in [2,9,14,20,23]:
3          try:
4              isnumber = float(lines[i])
5          except ValueError:
6              if i==9 or i==14:
7                  lines[i] = '0000'
8              else:
9                  lines[i]='0.00'
10
11     crs_dep_time = int(lines[9][0:2])*60 + int(lines[9][2:])
12     crs_arr_time = int(lines[14][0:2])*60 + int(lines[14][2:])
13     data_row = [lines[1],lines[2],lines[3],lines[4],lines[5],lines[6],lines[7],
14                 lines[8],crs_dep_time,crs_arr_time,lines[20],lines[23],lines[11],lines[16]]
15     return(data_row)
16
17 if __name__ == "__main__":
18     input_path = "/group03/Predict-Delay/Dataset/whole_train_data.csv"
19     output_path = "/mnt/wiktorskit-jungwonseo-ns0000k/home/notebook/group03/Renny-temp/testing"
20
21     if not os.path.isfile(input_path) or "/mnt/" not in input_path or len(output_path)==0:
22         print("Please check your input path again")
23         sys.exit(-1)
24
25     sc = pyspark.SparkContext.getOrCreate()
26
27     text_file = sc.textFile("file://"+input_path)
28     counts = text_file.map(lambda line: line.replace('"', '').split(',')) \
29                   .filter(lambda line: len(line) >= 29 and line[0] != 'YEAR' and line[16] != ''
                       ↪  and line[11] !='') \
30                   .map(convert)
31
32     counts.saveAsTextFile(output_path)
33     sc.stop()
```

Listing 2: Pre-processing Spark code

Code implementations can be seen in listing 1 for MRJob and listing 2 for Spark. As seen in the code in listing 1, line 5-6 is to remove the (") character, split the line by comma, and strip the words from whitespace. Then the data is filtered in line 7, we only take the row which departure delay and arrival delay not empty. Line 8 - 15 is for handling and filling missing or incompatible value for the numerical columns, as it will return error in the next stage if the value is nan or string. Line 17-18 is to convert the departure and arrival time from time (hhmm) format to number of minutes (relative to 00:00). Line 20, is to put together the item in a list. line 22, to yield the lines with 'null' key. The sequence of code implementation is similar for Spark.

After the pre-processing steps, the cleaned data is stored in hdfs for MRJob and RDD for Spark to be used for the next algorithm. We run the pre-processing to train and test dataset separately, and also save them as two files.

3 Algorithm and Implementation

3.1 Algorithm

We choose Random Forest to build our predicting model since it can parallelize well and we could run them on Apache Hadoop or Spark to deal with the big data. Random forest is

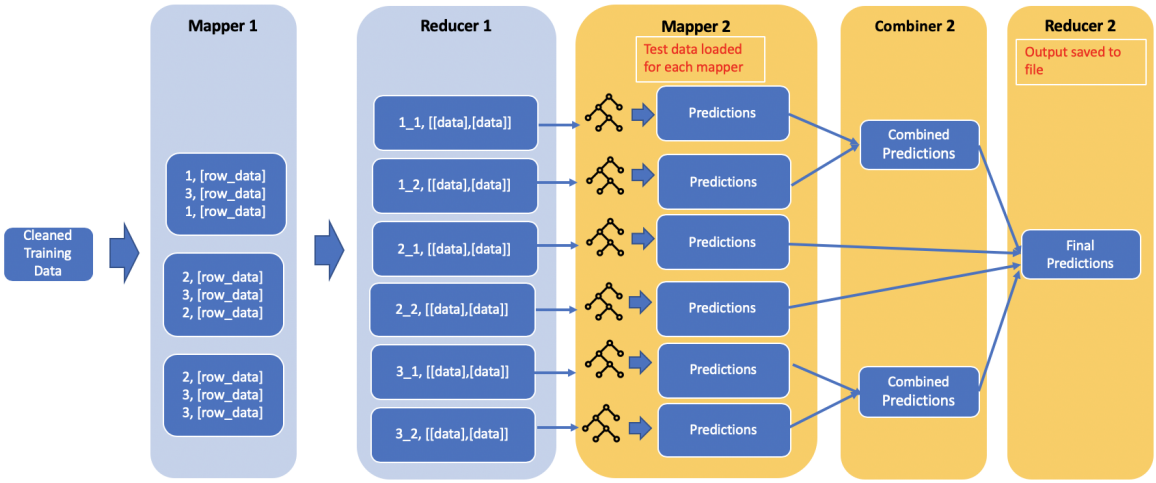

Figure 3: Our random forest prediction model implementation

an ensemble learning method that's constructed from multiple decision trees and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees[8][9]. In this project, we try to solve a regression problem as predicting delay time. Therefore, we need to calculate the mean value from multiple trees predictions to get the final predicted values.

We build random forest in three different methods then compare them together on running time and accuracy:

- **MRJob-** build from scratch and run on Apache Hadoop (sandbox).
- **PySpark-** build from sratch and run on Spark cluster.
- **MLlib-** use library and run on Spark cluster.

3.2 MRJob Implementation

The structure and workflow of our implementation in MRJob are demonstrated in figure 3. Although this diagram is more specific to MRJob implementation, our PySpark work also follows the same idea.

```
1  def configure_options(self):
2      super(MRPredictDelay, self).configure_options()
3      self.add_passthrough_option('--maxSplitNumber', type='int', default=10)
4      self.add_passthrough_option('--sampleNumber', type='int', default=100)
5      self.add_passthrough_option('--sampleSize', type='int', default=4000)
6      self.add_passthrough_option('--predict', default='departure', choices=['departure',
       ↪  'arrival'])
7      self.add_file_option('--testFile')
```

Listing 3: Configure options in MRJob class

At first, we configure options for the algorithm to allow us to change the algorithm parameters when we run the command (listing 3). The options consist of:

- **maxSplitNumber-** maximum number of splits from the input training data.
- **sampleNumber-** number of subsample we get when we do sampling (without replacement) on a single split.

```
def steps(self):
    return [
        MRStep(mapper = self.mapper_prepare_data,
                reducer = self.reducer_prepare_data),
        MRStep(mapper_init = self.mapper_init_predict,
                mapper = self.mapper_predict,
                combiner = self.combiner_predict,
                reducer = self.reducer_predict)
    ]
```

Listing 4: Define steps in MRJob class

- **sampleSize-** number of flight records in a subsample.
- **predict-** either *arrival* or *departure*. If it's set to *arrival*, then the program will predict the delay time on arrival and vice versa.
- **testFile-** path to the test file in HDFS.

Then, we define the steps as in listing 4. Our MRJob class has two steps as described in the following details:

- **Step 1 - Mapper:** Splitting randomly the data into maximum *maxSplitNumber* parts by generating key as a random number between 0 and *maxSplitNumber*, value is parsed from the input line of the training data file. The main code logic of this mapper is described in listing 5.

```
key = random.randint(0, self.options.maxSplitNumber)
line = line.replace('"', '').split('\t')
record = line[1].split(',')
yield key, record
```

Listing 5: MRJob implementation, Step 1 - Mapper

- **Step 1 - Reducer:** with each data split, we generate *sampleNumber* make subsamples randomly, each subsample contains *sampleSize* flight records and has been sampling without replacement. The code is listed in listing 6. This reducer yields the key as a combination of split number and subsample number in the format "{}_{}", and value is a subsample.

```
l = values_list.shape[0]
# sampling without replacement
for i in range(0, self.options.sampleNumber):
    idx = np.random.choice(l, size=self.options.sampleSize, replace=False)
    yield "{}_{}".format(key, i), values_list[idx, :].tolist()
```

Listing 6: MRJob implementation, Step 1 - Reducer

- **Step 2 - Mapper Init:** in the second step, we will build a decision tree for each subsample. All decision trees need to have the same test dataset in mapper, so in this mapper init we load the test data from test file in HDFS and store into a class attribute *self.test_set*.

- **Step 2 - Mapper:** run decision tree model and get predictions. Code is listed in listing 7. We separate decision tree in a different class, its code is a bit long, so we don't mention it here and you can find it from git repository.

```
def mapper_predict(self, key, trainingset):
    # create a decision tree
    DT = DecisionTree(training_data = trainingset, field_types = self.field_types)
    model = DT.build_model()
    predictions = DT.predict(model, self.test_set)
    yield 'predictions',(1,predictions)
```

Listing 7: MRJob implementation, Step 2 - Mapper

- **Step 2 - Combiner:** we add combiner to the second step to speed up the performance. In each combiner, we calculate number of predictions come to that combiner task and also sum all predictions into one combined predictions then yield them to the reducer. Code for this part is described in listing 8.

```
def combiner_predict(self, key, predictions):
    predictions = list(predictions)
    combined_prediction = []
    predictions_number = len(predictions)
    for i in range(0, predictions_number):
        if i == 0:
            combined_prediction = predictions[i][1]
        else:
            combined_prediction = np.add(combined_prediction, predictions[i][1])
    yield key, (predictions_number, combined_prediction.tolist())
```

Listing 8: MRJob implementation, Step 2 - Combiner

- **Step 2 - Reducer:** we calculate the total number of predictions and sum the predictions from all combiners and mappers, then calculate the average values correspond to each record in test set and that is the final predictions.

```
def reducer_predict(self, key, predictions):
    predictions = list(predictions)
    final_prediction = []
    predictions_number = 0
    for i in range(0, len(predictions)):
        predictions_number += predictions[i][0]
        if i == 0:
            final_prediction = predictions[i][1]
        else:
            final_prediction = np.add(final_prediction, predictions[i][1])

    # calculate the average from all predictions
    final_prediction = final_prediction/predictions_number

    yield key, final_prediction.tolist()
```

Listing 9: MRJob implementation, Step 2 - Reducer

3.3 Pyspark Implementation

In Pyspark implementation, we follow the same logic as we did with MRJob. Main part of this implementation is listed in the listing 10. We load the training data and test data from HDFS files into RDDs then follow the chain of transformations and actions as below:

- **map-** get data from a line.
- **map-** generate a random number as a key so we can split the training data into *maxSplitNumber* splits. *removeNonPredict* is just a function to remove either "arrival delay time" or "departure delay time" depend on what target we want to predict.
- **reduceByKey-** merge all the records which have same key into one list using *numpy.extend*.
- **flatMap-** make subsamples on each data split using the similar code as Step 1 - Reducer of MRJob implementation (listing 6). We use *flatMap* instead of *map* transformation since we need to return (key, value) pairs for multiple subsamples.
- **map-** get predictions from the decision trees. We write decision tree code in a separate class so we can re-use it in both MRJob and PySpark implementations.
- **reduce-** action to calculate the total number of predictions and sum of all predictions.

```
def reduceSplit(x, y):
    x.extend(y)
    return x

def samplingData(key, values):
    values_list = np.array(values)
    l = values_list.shape[0]
    if l < sampleSize:
        return [(key, values)]
    else:
        # sampling without replacement
        pairs = []
        for i in range(0, sampleNumber):
            idx = np.random.choice(l, size=sampleSize, replace=False)
            pairs.append(("{}_{}".format(key, i), values_list[idx, :].tolist()))
        return pairs

def predict(training_set, test_set):
    DT = DecisionTree(training_set, field_types)
    model = DT.build_model()
    predictions = DT.predict(model, test_set)
    return predictions

result = (training_rdd
            .map(lambda line: line.replace("[","").replace("]","").replace("'","").split(','))
            .map(lambda line: (random.randint(0, maxSplitNumber), [removeNonPredict(line,
            ↪  predict_on)]))
            .reduceByKey(reduceSplit)
            .flatMap(lambda record: samplingData(record[0], record[1]))
            .map(lambda training_set: (1, predict(training_set[1], test_set)))
            .reduce(lambda x, y: (x[0] + y[0], np.add(x[1], y[1]))))

final_predictions = result[1]/result[0]
```

Listing 10: Chain of transformations and actions in PySpark implementation

3.4 MLlib Implementation

The MLlib implementation is using random forest library for regression. spark.mllib implementation of random forests is built on top of the existing decision tree implementation[4]. Each tree is trained independently in parallel using subsample of training data and subset of features. The sampling and subset of data is done randomly in the implementation.

Before building the trees, we need to prepare the training and test data and convert them to the LabeledPoint datatype, which is required for the RandomForest.trainRegressor method. The data preparation code can be seen in listing 11. Line 2-9 is to convert train and test data to Dataframe so we can apply stringIndexer method in the next step to change categorical columns to numerical category. LabeledPoint datatype can only receive number as it's input, therefore we need to convert all categorical features which in string format to numerical category format. We initiate this step by defining which columns are numerical and which columns are categorical features (line 11-12) and then pass it to the StringIndexer method in (line 17-19). StringIndexer encodes the string value to indices which is sorted by the frequency. VectorAssembler in line 21-22 is a transformation to combine list of all feature columns into one vector column. If you notice, we joined the train and test data in line 8 so when we use stringIndexer to encode the numerical index, we covers all available categories in each feature. After transforming the dataframe with above changes(line 27), we split again the data to train and test datasets by applying filter(line 29-30), and convert the train and test datasets to LabeledPoint data type using map operation on each row by passing the label (the target value) and vector of features.

After data preparation, we feed in the LabeledPoint training data to the randomForest.trainRegressor method. The function can be seen in listing 12. Parameters that we can tune:

- **numTrees-** Number of trees in the forest.
- **featureSubsetStrategy-**Number of features to use as candidates for splitting at each tree node [6].
- **maxDepth-** Maximum depth of each tree in the forest [6].
- **impurity-** measure of the homogeneity of the labels at the node. The current implementation provides two impurity measures for classification (Gini impurity and entropy) and one impurity measure for regression (variance) [6].

For the prediction part, can be seen in list 13.

4 Accuracy Calculation

The Accuracy measure that we use in this project is Root Mean Squared Error (RMSE).

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\Big(\frac{d_i - f_i}{\sigma_i}\Big)^2} \tag{1}$$

The implementation is simple in both Hadoop and PySpark. In Hadoop it is done with 1 job consist of mapper, combiner and reducer, while in PySpark and MLlib we just make a mathematical function in the same file as the random forest implementation.

5 Performance Comparison and Tuning

In this section, we compare the running time and RMSE for all of the implementations in environments with different configurations. For Hadoop, we only have single configuration as we are using our local sandbox environment. We will also describe the tuning that we have implemented to make the code runs more efficient.

```
def get_train_test(file_path,library,predict):
    raw_train = sc.textFile("file://"+file_path[0])
    train_conv = raw_train.map(lambda x : convert(x,'train',predict))

    raw_test = sc.textFile("file://"+file_path[1])
    test_conv = raw_test.map(lambda x : convert(x,'test',predict))

    raw_all = sc.union([train_conv, test_conv])
    df = sqlContext.createDataFrame(raw_all, schema = ['label',
     ↪  '0','1','2','3','4','5','6','7','8','9','10','11','data_group'])

    numerical_features = ['1','8','9','10','11']
    categorical_features = ['0', '2', '3', '4', '5', '6', '7']

    stages = []

    #stringIndexer and save the stages for pipeline
    for categoricalCol in categorical_features:
        stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol +
         ↪  "Indexer")
        stages += [stringIndexer]

    assemblerInputs = [x + "Indexer" for x in categorical_features] + numerical_features
    assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
    stages += [assembler]

    pipeline = Pipeline(stages=stages)
    pipelineModel = pipeline.fit(df)
    df_model = pipelineModel.transform(df)

    df_train = df_model.filter("data_group=='train'").select('label','features')
    df_test = df_model.filter("data_group=='test'").select('label','features')

    data_LP_train = df_train.rdd.map(lambda x: LabeledPoint(x[0], x[1].toArray()))
    data_LP_test = df_test.rdd.map(lambda x: LabeledPoint(x[0], x[1].toArray()))

    return data_LP_train,data_LP_test
```

Listing 11: Converting data to LabeledPoint type

5.1 Using Combiner

The use of combiner have a significant effect in the efficiency, the running time on Hadoop was reduced from 42 minutes to 34 minutes with combiner. This is due to reduced number of key value pairs transferred from mapper to reducer that resulted in less traffic in the network (table 1).

5.2 Performance Comparison

5.2.1 MRJob, Pyspark, and MLlib comparison

In this section, we compare the implementations of random forest in MRJob-Hadoop, Pyspark and MLlib . Please note that the comparison is done on the basic settings for each environment, this will not be an apple to apple comparison, as we have different environment for Hadoop and spark. We use sandbox in our local computer for Hadoop-MRJob, and spark cluster provided by the University for Pyspark.

The result in table 1 was using 17 million flight records as the training dataset, and 2000 flight records for the test dataset with below parameters:

```
def mllib_random_forest(max_depth,num_trees,train_set):
    #Random Forest Regressor implementation
    model_rf = RandomForest.trainRegressor(train_set, categoricalFeaturesInfo={},
                                           numTrees=num_trees, featureSubsetStrategy="auto",
                                           impurity='variance', maxDepth=max_depth)
```

Listing 12: Build Random Forest MLlib

```
predictions = model_rf.predict(test.map(lambda x: x.features))
labelsAndPredictions = test.map(lambda lp: lp.label).zip(predictions)
```

Listing 13: Run Prediction MLlib

- **MRJob and PySpark-** maxSplitNumber = 1000, number of sampling = 3, sample size 4000. This gives us 3000 trees and each tree has 4000 training data.
- **MLlib-** numTrees : 10, maxDepth– 5, impurity='variance. We tried to run for higher number of trees (numTrees $\geq$ 100) in the basic settings but it failed due to the limited resource. We suspect that the sample train data for each tree in MLlib has a lot more data than our implementation in MRJob and PySpark (where we limit the number of train data to 4000 for each tree), therefore, if we try to use 3000 trees in MLlib it will be too much to handle.

The running time in Spark is generally faster from MRJob Hadoop which could be caused by two things. First, by running Hadoop in local sandbox, we only have 1 node while the basic setting in spark cluster have 2 workers so it is better in supporting parallel job. Second, spark does its processing in the main memory of the nodes which minimizes the operations to and from the disks[7].

The RMSE is quite consistent between 27-28 minutes for all implementations. Even though we run MLlib with only 10 trees, the RMSE is still slightly better than the rest which run with 3000 trees. This can be because MLlib use much higher number of sample train data in each tree than our from-scratch implementation in Hadoop and Pyspark which limits the number of sample train data to 4000 for each tree.

5.2.2 Pyspark and MLlib comparison

We ran the codes in different settings with specifications as table 2, while the master node always has same specification of 2 CPU and 2 GB Ram.

In figure 4, we can see that for both PySpark and MLlib, the running time generally decrease when we add number of workers or increase the specification for cpu and ram. However, if we see last two items, we can see that it is more efficient to use 2 workers with higher specification of cpu and ram (8 cpu and 8 GB ram) than using high number of workers (8 workers) with lower specification of cpu and ram (2 cpu and 2 GB ram). In our opinion, this can be caused by less network traffic when we only use 2 workers compare to 8 workers while in overall both configurations have almost the same resources in total.

Parameter	MRJob (with combiner)	MRJob (without combiner)	Pyspark	MLlib
Runtime (min)	34	42	25	23
RMSE (min)	28	28	28	27

Table 1: Comparison of MRJob, Pyspark and MLlib

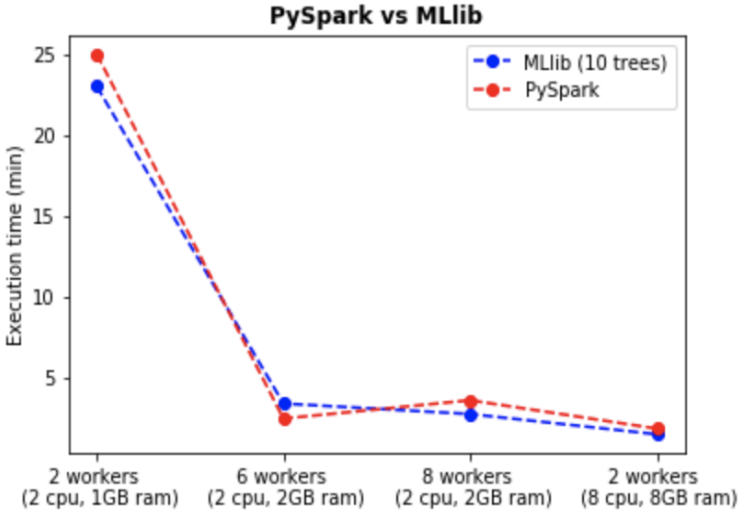

Figure 4: Comparison of MLlib implementation and Spark

Number of workers	2	6	8	2
Number of cores	2	2	2	8
Memory (GB)	1	2	2	8

Table 2: Test Environment specification

For the RMSE, MLlib implementation resulted in 27.74 min in average, while spark implementation resulted in 30 min in average.

6 Challenges

In the implementation of random forest from scratch, the biggest challenge is to find effective way to split and subsample training data as well as building parallel decision trees in map reduce architecture. Initially we only split the data to number of partitions and build tree for each partitions in parallel, however we continuously got error messages due to resource limitations. We suspect it is due to the data is to big. We fixed this issue by splitting the data, take subsamples from the splits and build the tress from the subsamples. We made that the parameters for the number of split, random sampling, and the size of sample can be adjusted.

We also experienced issue with data type in the beginning. When the data is transferred from mapper to reducer, it is in form of unicode, which gives error when we are implementing numpy operations. We found out that we need to convert them to numbers(i.e. float, int) before applying numpy operations.

7 Conclusion

Flight delay prediction is a good model for both airlines commercial players and customer to give insights for better planning and handling, which can potentially lead to cost savings. As the flight data is massive, we implemented random forest algorithm using distributed system such as hadoop or spark framework. Random forest is well known as a robust algorithm which is stable and less prone to overfitting, and also it is naturally support paralellism as it is an ensemble of decision trees that can be build independently of each other in parallel. By combining these characteristic with implementation on Hadoop or Spark framework, it can be a powerful tool for this task. For further work, there are still many ways to improve efficiency such as applying data balancing technique to train models better, adding more data such as weather forecast, or

using grid search [10] to optimize the hyperparameters for a better model accuracy.

References

[1] U.S. Passenger Carrier Delay Costs, Airlines for America, http://airlines.org/dataset/per-minute-cost-of-delays-to-u-s-airlines/

[2] Bureau of Transportation Statistics, https://www.transtats.bts.gov/OT_Delay/ot_delaycause1.asp?pn=1

[3] Bureau of Transportation Statistics, https://www.transtats.bts.gov/DatabaseInfo.asp

[4] Random Forests and Boosting in MLlib, Databricks, Jan 21, 2015, https://databricks.com/blog/2015/01/21/random-forests-and-boosting-in-mllib.html

[5] Decision Trees - RDD-based API, Apache Spark, https://spark.apache.org/docs/latest/mllib-decision-tree.html

[6] Ensembles - RDD-based API, Apache Spark, http://spark.apache.org/docs/latest/mllib-ensembles.html

[7] Micah Williams, Apache Spark vs. MapReduce, DZone, Aug 30 2017, https://dzone.com/articles/apache-spark-introduction-and-its-comparison-to-ma

[8] Ho, Tin Kam (1995). Random Decision Forests (PDF). Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14{16 August 1995. pp. 278{282

[9] Classification and Regression Trees Leo Breiman, Jerome Friedman, Charles J. Stone, R.A. Olshen (1984)

[10] Bergstra, James; Bengio, Yoshua (2012). "Random Search for Hyper-Parameter Optimization". Journal of Machine Learning Research. 13: 281{305.