# Project: Traffic Prediction

Milan Lagae

Institution/University Name: Vrije Universiteit Brussel
Faculty: Sciences and Bioengineering Sciences
Course: Big Data Processing

## Contents



## 1 Intro

This report will discuss an implementation for the assignment "Project: Traffic Prediction" for the course: Big Data Processing. First, the implementation itself will be discussed in section §2. Following that, answers to the required questions in section §3. And lastly, a small section on performance benchmarks in section §4.

## 2 Implementation

This section will discuss the implementation (code) for the project. Full project code can be found in the associated Apache Spark project or small snippets will be placed in the text or larger ones in the Appendix section §5.

### 2.1 Overview

All the files can be found in the traffic package of the bdp-traffic folder. The traffic package consists of the following files:

- Traffic.scala
- TrafficLoader.scala
- TrafficJoiner.scala
- TrafficTimeSeries.scala
- TrafficTransformer.scala
- TrafficPredictor.scala

The files are structurd in the ordered in which they are applied to the input.Each file also has its own logger variable set, which is used for logging. For this, the build.sbt file was modified with an additional package. Each class in the pipeline receives a reference to the SparkSession by the spark variable.

### 2.2 Traffic.scala

This is the file that is executed when the project is ran. It executes the different steps (files) in pipeline manner. The complete execution pipeline can be seen in listing: Figure 1.

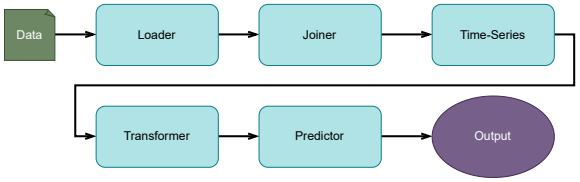


Figure 1: Execution Pipeline

### 2.3 Loader

The loader or TrafficLoader.scala file is responsible for loading the correct data set files. For this 3 different methods are created for each type of file: loadVolume, loadSpeed and loadFeatures. Loading the correct file is done based on the dataset value which is an enum, as shown in listing

```
object DataSet extends Enumeration {
  type DataSet = Value

  val L_Tiny, M_Tiny_Select = Value
}
```

This way, if different datasets are required to be tested this can be easily extended. In the Traffic.scala file, each dataframe is seperately loaded and stored in a dataframe.

### 2.4 Joiner

In this step of the pipeline, the 3 dataframes are unpivotted & joined together to create one big dataframe, for application of time-series values in next step.

#### 2.4.1 Features

This step in the pipeline is responsible for joining the different data files in one melted dataframe.

First, start by adding an id column to the features dataframe. For this the RDD of the dataframe is accessed. This is done using a zipWithIndex and a map. This ensures a consisting numerically ordered id, over different partitions [1], [2]. After adding the id column, the RDD is transformed back into a dataframe.

#### 2.4.2 Speed & Volume

The following operations are identical for both dataframes: speed & volume. Each dataframe is unpivotted, from a wide format to a long format. For this, the sql method stack is used.

After the respective dataframe is unpivotted, the node column is updated to a int type, by removing the node_ from the column name and casting it to an int value. This allows the column to be used as a feature when applying the Vector Assembler.

#### 2.4.3 Joining

The speed & volume dataframes are joined on the node & timestamp columns, for an inner join operation. From the features dataframe, a select list of columns is selected based on assumption of relevancy for the prediction model.

Finally, the selected features dataframe is joined with the earlier speed & volume dataframe on node column value.

### 2.5 Time-Series

This section will describe which time-series features were added to the dataframe.

#### 2.5.1 Lag

For both speed & volume, the following lag features were added:

- half hour/ 30 minutes = 6 rows
- 1 hour / 60 minutes = 12 rows
- 1 day / 1440 minutes = 288 rows

The default value chosen for the lag is 0, since this value works better with the prediction models in mllib.

#### 2.5.2 Window

A rolling window of half an hour and 1 hour was added for both the speed & volume metric.

At the end of the updated dataframe, the operation to replace all null values with zero is applied, this prevents any issues with the prediction model in subsequent steps.

### 2.6 Transformer

This step of the pipeline will apply the Vector Assembler to the selected columns to create a features column to the given dataframe. This done once for the historical training data and multiple times during the prediction phase for the test data.

For this reason, there are 2 different transform methods contained in the class.

### 2.6.1 Training Data

Preparing the given dataframe for the training phase is done with the InitTransform method. The features column is generated by applying the VectorAssembler to the selected list of columns marked as featured. The resulting dataframe is called: transformedDF.

To prematurely optimize, the historical dataframe is selected for the most important values. To start with, the training dataframe (trainingDataDF) is created, by selecting the columns: node, timestamp, speed, volume and features.

Furthermore, the combination of each node and it's selected list of static features is selected from the transformedDF dataframe, to which the distinct operation is applied, creating the unqique combination of each node with it static features (nodeFeaturesDF).

To further reduce the number of rows required to be kept in memory during the prediction phase, a select number of rows, 287 to be specific is selected from the transformedDF by the takeHistory method. The number of rows is specific to the maximum number of rows requires for the time-series step. Shifting the rows by 1 day (288 rows).

Taking the 287 latest row for **each** node, is done by first generating a Window, which is parititioned on the node column and ordered descendingdly (latest timestamp first) on the timestamp. Selecting the correct rows is done by adding a new column rn, to which the previously made Window is used for generating the correct row number. The rows are than filtered on the column number, dropping any for which the following holds: $rn > 287$. Lastly, a select is performed to select the required list of columns, following the transformation.

Following the transformation of the input data, and generated baseDataDF dataframe, the latest timestamp in the historicaldata is retrieved.

A tuple of 4 values is returned from the prepartion transformation step: baseDataDF, trainingDataDF, nodeFeaturesDF, endTime.

### 2.6.2 Predicting Data

For the transformation step during the prediction phase of the pipeline, the VectorAssembler is again applied on the input dataframe and features column is generated. A single value: transformedDF is returned.

## 2.7 Predictor

A flowchart visualiation of the steps taken during predictor phasse of the pipeline can be found in Figure 2, a legend is available in Figure 5.

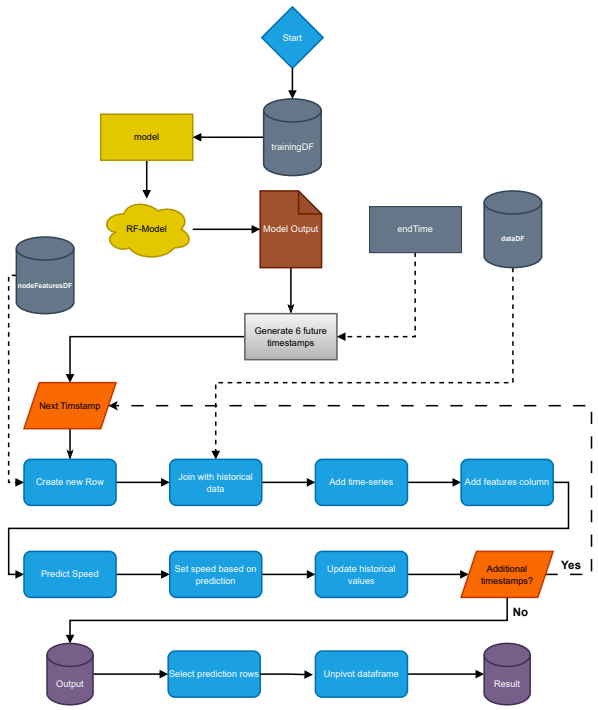


Figure 2: Prediction Step Flowchart

The RandomForestRegressor model is created, with the label column: speed & features column: features. Other values are left default. The model is fitted on the training dataframe (trainingDF). The generated model is written to file as described in the assignment.

From the given latest timestamp (endTime), 6 future values are generated. The list of future timestamps is iterated.

During iteration, the additional rows are created by adding the timestamp, speed and volume columns to the nodeFeaturesDF. The future rows are added to the historical dataframe (dataDF) by using unionByName. The time-series columns are created by applying the addTimeSeries method on the dataframe. Once the time-seriese columns have been added, the features columns is generated.

With the prepartion phase complete, the prediction can be done by applying the transformedDF to the rf_model. The result dataframe contains a prediction column, containing the prediction for the speed of each node on that particular timestamp.

To include the predicted speed value in further prediction of speed for time-series features, the rows containing the current timestamp have their speed (originally 0) set to the value in the prediction column (predicted value).

Lastly, the historically data is updated, by removing one row from the original dataset and including the new row with predicted speed, by applying the takeHistory method on the updateDF dataframe. The result of this application is put in the mutable variable: dataDF, which is continuously updated during the iteration.

After the predictions, the predicted rows are selected by filtering out all the rows that have a timestamp that is lower than the endtime. On the predictions dataframe, the unpivot operation is applied, so the dataframe is in correct format for displaying information in the terminal.

### 2.8 Output

The result of the prediction is returned to the Traffic.scala file. The generated prediction data (predictions) is passed to the writeFile method, which is responsible for printing and (maybe) writing output to a file.

The generated data frame is iterated and the values for each timestamp are written per line, with values being the speed for each node. If required, a boolean: file can be set to write the output to a file.

## 3 Discussion

This section will discuss the answer to the questions formulated in the assignment.

### 3.1 Question 1

**Question**: Have you persisted some of your intermediate results? Why could persisting your data in memory be helpful for this pipeline?

Based on the benchmarks performed in section §4, the answer to this question is, that persisting data for this pipeline has a negative effect. Possible reasons for this is the fact, that the used dataset is quit small in comparison to the full dataset(s) available. A proper conclusion cannot be made without further testing.

### 3.2 Question 2

**Question 2:** In which parts of your implementation have you used partitioning? Did this impact performance? If so,why?

Based on the benchmarks performed in section §4, it can be concluded that for this implementation, changing the default partitioning values, has a considerable impact on performance, details can be found in full in the section §4.

### 3.3 Question 3

**Question 3**: Which datastructure(s) does your implementation use: RDDs, DataFrames, or Datasets? Please motivate your choice.

The implementation makes mostly use of the dataframes, since these, as seen in clase have the best performance optimization enabled under the hood. For reasoning on why RDD's were used in one specific section please see: §2.4.1.

### 3.4 Question 4

**Question 4:** Which predictive algorithm did you use and why?

The chosen predicate model is: RandomForestRegressor, since this is what was recommend in the FAQ section of the assignment.

# 4 Benchmarks

For all benchmarks, 4 runs were done. The first run was considered a dry run, while for the proceeding 3, the average was taken. For time constraints reasons, the largest dataset: M_5000 was not tested.

## 4.1 Specifications

The specifications of the devices used for the benchmarking can be found in the section §5.

## 4.2 Results

This section will discuss the result of the benchmarks performed on the given datasets. Configuration for each benchmarking scenario can be found in Table 1.

### 4.2.1 Desktop vs Macbook

This comparison will discuss the difference in performance between the Macbook in Table 2 and desktop in Table 3.

**Result**

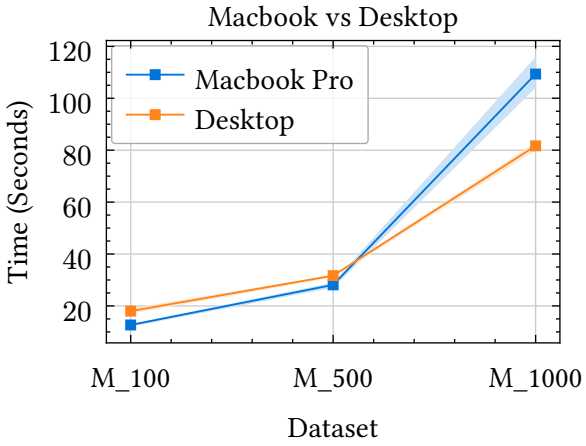


Figure 3: Desktop vs Macbook Comparison

### 4.2.2 Partitioning

This comparions will discuss the affect the number of paritions has on the performance of the implementation.

**Result**

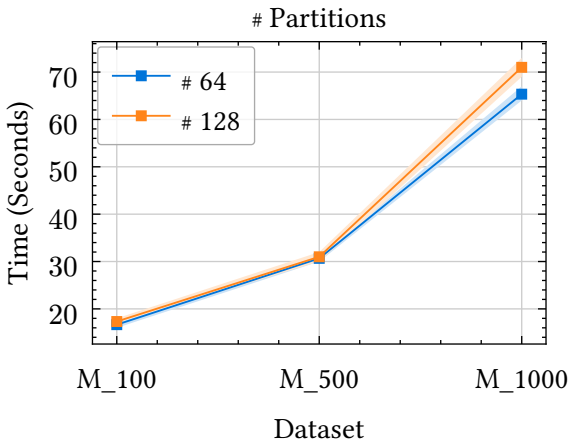


Figure 4: Number of Partitions Comparison

### 4.2.3 Conclusions

# 5 Appendix

## 5.1 Benchmarks

### 5.1.1 Benchmark Scenarios

| Scenario | Branch | # Cores | Memory (driver & Executor) | # Partitions | Partition Size |
|---|---|---|---|---|---|
| Desktop vs Macbook | Local | 6 | 10 | 64 | 128 |
| # Partitions | Local | 16 | 24 | Dynamic | 128 |

Table 1: Benchmarking Scenario Settings

### 5.1.2 Macbook

| Part | Value |
|---|---|
| CPU | M2 Pro (6 performance and 4 efficiency) |
| RAM | 16GB |
| OS | MacOS 15.7.2 (24G325) |

Table 2: Macbook Specifications

### 5.1.3 Desktop

| Part | Value |
|---|---|
| CPU | Ryzen 9 5950X |
| RAM | 64GB (3200Mhz) |
| OS | Windows Versie 10.0.22631 Build 22631 |

Table 3: Desktop Specifications

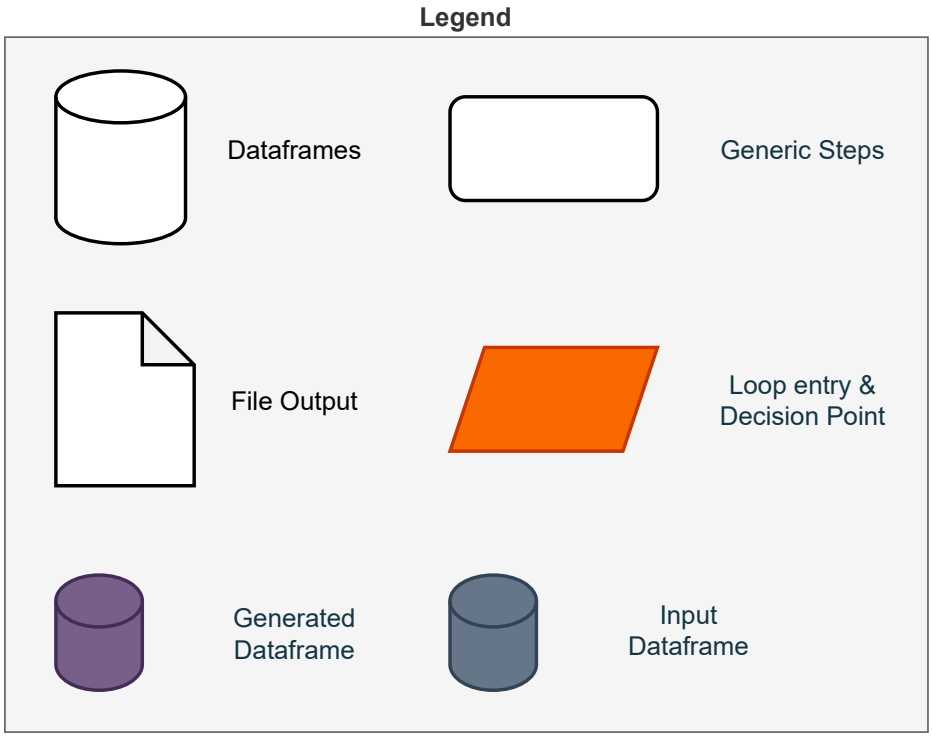


Figure 5: Prediction Step Flowchart Legend

## Bibliography

[1] [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.monotonically_increasing_id.html

[2] [Online]. Available: https://stackoverflow.com/questions/35705038/how-do-i-add-an-persistent-column-of-row-ids-to-spark-dataframe