# 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.

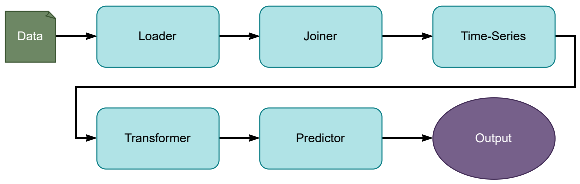



Figure 1: Execution Pipeline

### 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, so the model can be trained on it.

The Vector Assembler is first applied to a sequences of selected columns, with the output name of the column being: features. After the features column is generated, additional rows for each node are added.

The oldest timestmap from the existing dataframe is retrieved an parsed into the correct Java type. Starting from the oldest timestamp, 6 additional timestamp values are generated by manner of a map and cast to a dataframe consisting of 6 rows and a single column named: timestamp.

Proceed to select the columns node & features and apply the distinct operation.

The two dataframes consisting of the distinct nodes & features & timestamps are joined together by using the crossJoin operation. For each distinct relation of a node & feature, 6 additional timeestamp rows are created. One additional row named: speed is added with default value of: 0.

From the dataframe vector, select the columns: timestamp, node, speed and features. Proceed to apply the unionByName operation on this dataframe and the previously generated dataframe. This concludes the steps for preparing the data for model prediction.

Return the final dataframe and largest timestamp from the original data as a tuple, will be used in the prediction step.

### 2.7 Predictor

The data can now be split in training data and 'test' data. The data datafarme is split by applying a filter on the value of the timestamp column. All rows with an equal or lower timestamp value are considered training data, while larger timestamps are the prediction data.

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 (trainDF). The generated model is written to file as described in the assignment.

Lastly, the model is applied to the prediction dataframe (predictionDF) to predict future speeds.

### 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

### 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?

**TODO**

### 3.3 Question 3

**Question 3**: Which datastructure(s) does your implementation use: RDDs,DataFrames,orDatasets? 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 the types of benchmarks ran on each host on the local context, both used the same provided dataset:

- Type 1: No cache/persist & other default settings
- Type 2: Cache & other default settings
- Type 3: Persist & other default settings

### 4.1 Specifications

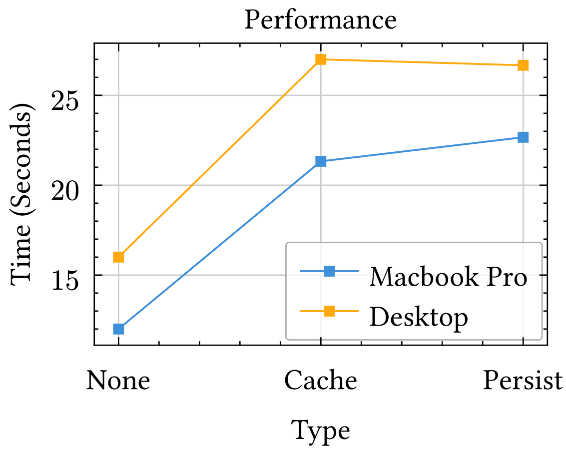
#### 4.1.1 Macbook

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

#### 4.1.2 Desktop

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

### 4.2 Results



## 5 Appendix

## 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