Flight Delay Forecasts: Developing a Predictive System Using Weather and Traffic Data

Slim Kachkachi, Arij Sakhraoui, Sara Zanutto

Executive Master's in Artifical Intelligence and Data Science - October 2023
Université Paris Dauphine

Contents

List of Figures

List of Tables

Introduction

Air travel, in the modern era, has emerged as one of the primary modes of transportation, connecting continents and cultures. As the frequency of air travel increases, so does the importance of ensuring a timely and efficient journey for passengers. However, flight delays remain a persistent challenge for airlines, causing inconvenience to travelers and leading to significant economic losses for the aviation industry. According to the Federal Aviation Administration, flight delays in the U.S. alone cost approximately \$8 billion annually. The impact is not just monetary; delayed flights can result in missed connections, overnight stays, and a cascade of disruptions affecting various stakeholders.

Predicting flight delays, therefore, holds immense potential. Accurate predictions can empower airlines to optimize operations, communicate more transparently with passengers, and minimize the cascading effects of delays. Moreover, travelers can make informed decisions about their journeys, leading to a more satisfactory travel experience. From a broader perspective, effective delay prediction models can bolster the aviation industry's sustainability, enhancing its economic viability and environmental efficiency.

In this paper, we zero in on weather-induced flight delays. Utilizing Spark for data processing and leveraging machine learning models, we aim to understand how weather conditions influence flight timings. We explore the intricacies of building a predictive model, discuss the challenges faced, and delineate the results achieved.

1. Data preprocessing

1.1. Flight data

Among the datasets at our disposal to undertake our project, the first one includes the flight information for the year 2012 and 2013. The dataset presents the following fields:

- **FL_DATE:** The date of the flight (e.g., 2013-12-01).
- **OP_CARRIER_AIRLINE_ID:** A unique identifier for the airline carrier.
- **OP_CARRIER_FL_NUM:** The flight number.
- **ORIGIN_AIRPORT_ID:** The ID of the origin airport.
- **DEST_AIRPORT_ID:** The ID of the destination airport.
- **CRS_DEP_TIME:** The scheduled departure time.
- **ARR_DELAY_NEW:** The arrival delay in minutes.
- **CANCELLED:** A flag indicating if the flight was cancelled (0 = not cancelled, 1 = cancelled).
- **DIVERTED:** A flag indicating if the flight was diverted (0 = not diverted, 1 = diverted).
- **CRS_ELAPSED_TIME:** The scheduled elapsed time of the flight in minutes.
- **WEATHER_DELAY:** Delay in minutes due to weather (appears to have many missing values).
- **NAS_DELAY:** Delay in minutes attributed to the National Airspace System (NAS), also with many missing values.
- **Unnamed: 12:** An unnamed column with missing data, likely an artifact of the data collection or formatting process.

1.1.1 Flight data analysis

- **Variation of delayed flights per day of the week**

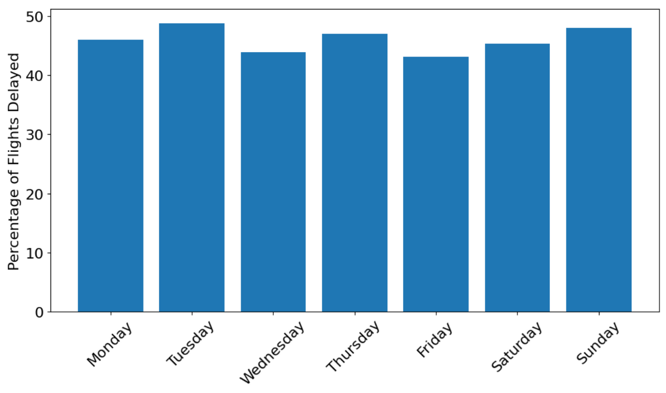


Figure 1: Weekly distribution of delayed flights

The percentage of delayed flights seems to be fairly uniform across the days of the week. Such uniformity suggests that external factors affecting flight delays, such as airport operations, airline practices, or weather patterns, might be consistent throughout the week. Alternatively, this could also indicate that various factors that cause delays might balance out over each day.

- **Variation of the percentage of delayed flights per hour of the day**

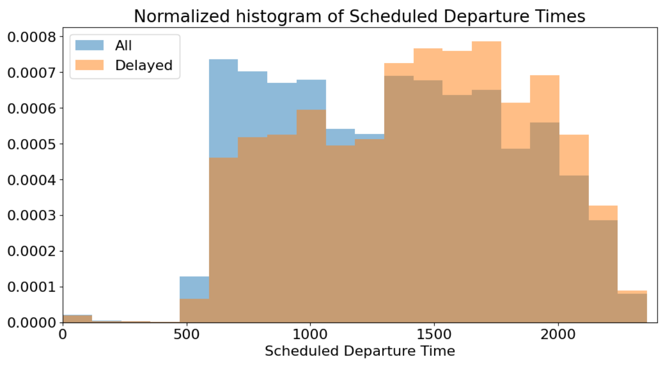


Figure 2: Normalized histogram of Scheduled Departure Times

From 1500 (3:00 PM) onward, there's an increase in the frequency of all scheduled flights, peaking around the 1700 (5:00 PM) mark. During this period, the difference between the total flights and the delayed flights seems to decrease, indicating that a higher proportion of flights tend to get delayed in the late afternoon and early evening.

- **Variation of the percentage of delayed flights per day of the month**

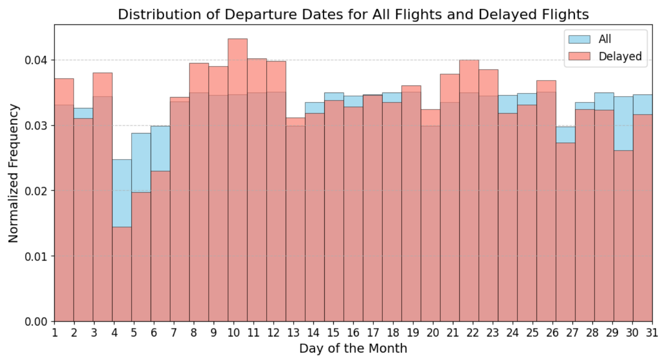


Figure 3: Percentage Variation of Delayed Flights by Day of Month

The distribution of delayed flights (represented by the red bars) shows some days with higher frequencies of delays than others. For instance, around the 9th, 18th, and between 24th to 28th, there's a noticeable peak in delayed flights.

- **Delays per aircraft**

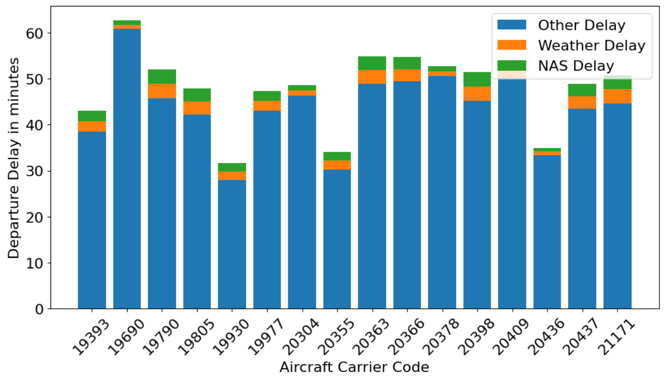


Figure 4: Delay Type Distribution per Aircraft

This shows that aircraft 19977 is the least likely aircraft to be delayed, and we can see the repartition of different causes of delays.

- **Attribute Selection and Potential Data Redundancies**

In our analysis of the Flights data set, we specifically chose to focus on four key attributes: "ORIGIN_AIRPORT_ID", "DEST_AIRPORT_ID", "CRS_DEP_TIME", and "ARR_DELAY_NEW". While we had the option to compute the real-time departure and arrival data for use as input during the Machine Learning phase, this could have inadvertently created a direct correlation between the target variable and the training data. We aimed to avoid this potential pitfall. Notably, there exists a degree of overlap in delay data between Weather delay and NAS delay attributes when compared to the arr_delay_new attribute. We tested the effect of this data leakage and we reported the result on the 'Baseline model' paragraph.

1.1.2 Flight data preprocessing

The flight data pre-processing involved a series of crucial steps aimed at preparing the dataset for the subsequent data transformation phase.
Initially, flights marked as either diverted or cancelled were removed from the dataset by utilizing the corresponding 'diverted' and 'cancelled' columns. Following this, these columns were removed as not necessary for the following steps.
Then, calculations were performed to determine the scheduled arrival time for each flight. This involved transforming the data into timestamps and summing the departure time to the "CRS_DEP_TIME."
However, the arrival times obtained in the previous step were not in local time. To rectify this, a table associated with time zones was joined, in order to allow the recalculation of arrival times based on respective local time zones. This adjustment ensured the correct alignment of the Flight table with Weather Observations in local time.

1.2. Weather observation data

In this subsection, we systematically examine the Weather Observation dataset. The intent is to refine the dataset beyond basic cleaning and structuring, optimizing it for sophisticated analytical processes. Through a methodical approach, encompassing statistical analysis, we endeavor to provide a detailed overview of meteorological data trends and irregularities. This rigorous analysis is pivotal, as it establishes the groundwork for subsequent, more granular investigations, and enhances our comprehension of weather's influence on flight dynamics. For the rest of this section, we will be working with the hourly 201201hourly.

1.2.1 Overview of the Weather data

This weather dataset, comprising 31,401 entries, offers a comprehensive view of various meteorological conditions. The data spans multiple parameters, essential for in-depth weather analysis and forecasting. Each entry is meticulously categorized into 45 distinct columns, ensuring a detailed representation of weather phenomena.
Key Data Points:

1. **Temporal Coverage**: The 'DateTime' column, a combination of date and time, provides precise temporal markers for each observation, essential for chronological analyses and trend identification.

2. **Temperature Data**: The dataset includes both Fahrenheit ('DryBulbFarenheit') and Celsius ('DryBulbCelsius') measurements, with over 31,200 valid entries for each.

3. **Sky Condition**: Annotated in 'SkyCondition', this parameter is crucial for understanding cloud cover and general sky visibility. It is a complex categorical type with over 200 categories, described with codes like OVC030 BKN025 OVC034.

4. **Visibility**: Recorded in miles, visibility data ('Visibility') is present in over 25,400 entries. It is is a numeric attribute with values ranging from 0 to 10,. An example value observed is 7.00, indicating moderate visibility. This aspect is vital for aviation, since it helps understand weather patterns like fog and mist.

5. **Humidity**: The 'RelativeHumidity' column, available in over 27,400 entries, provides insights into atmospheric moisture levels, impacting weather comfort levels and precipitation processes.

6. **Wind Characteristics**: Wind speed ('WindSpeed') and direction are crucial for weather prediction and understanding local climate behavior. The dataset contains wind speed data in over 26,700 entries.

7. **Precipitation**: Hourly precipitation data ('HourlyPrecip'), although available in only 289 entries, offers valuable insights into rainfall patterns, possibly impacting flight delays.

8. **Weather Type**: The 'WeatherType' is a complex categorical attribute that specifies weather conditions like fog, thunderstorms, etc..

1.2.2 Weather data analysis

- Wind Speed

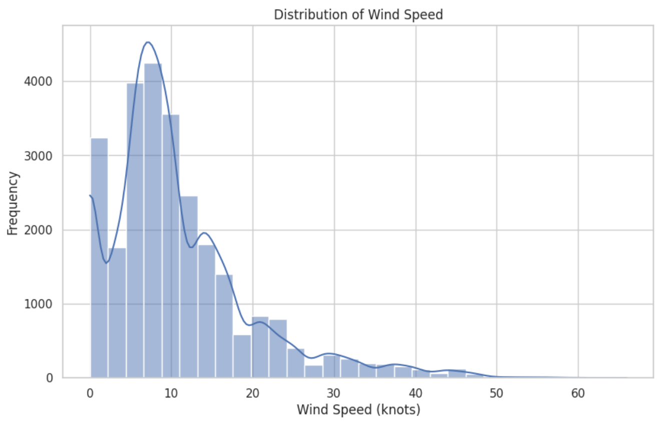


Figure 5: Distribution of Wind Speed

Wind speeds are mostly in the lower range, with a decrease in frequency as speeds increase, suggesting that high winds are less common and may not significantly impact flight operations during this month.

- Weather Types

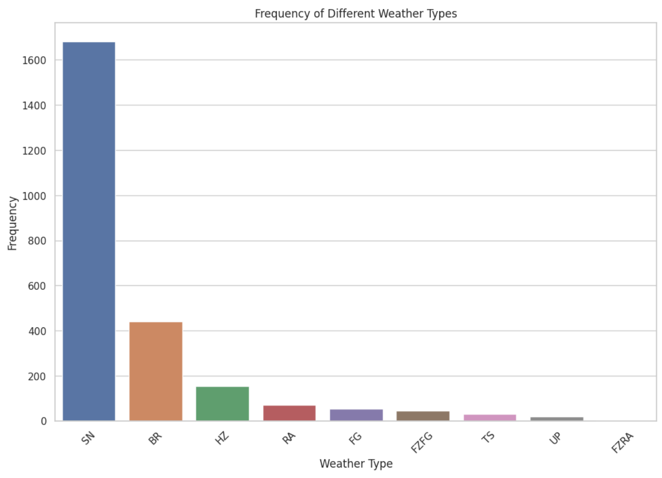


Figure 6: Frequency of Different Weather Types

This graph shows the frequency of various weather types, with snow (SN) being the most reported condition. The presence of fog (BR) and haze (HZ) could contribute to visibility issues affecting flight schedules

- Hourly Precipitation

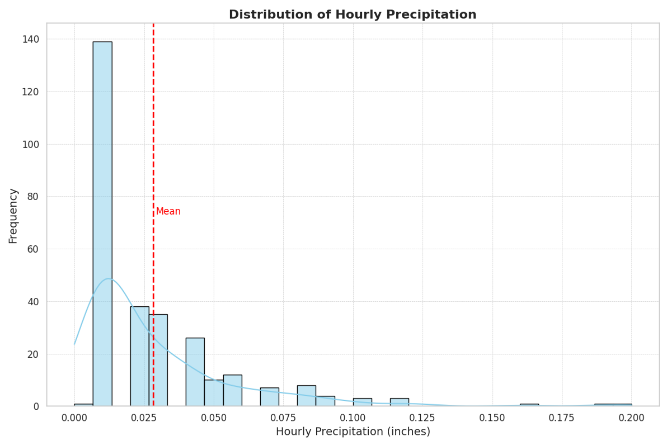


Figure 7: Distribution of Hourly Precipitation

Histogram showing the distribution of hourly precipitation levels. The low frequency of precipitation events indicates that rainfall was an infrequent occurrence during the month.

- Sky Conditions

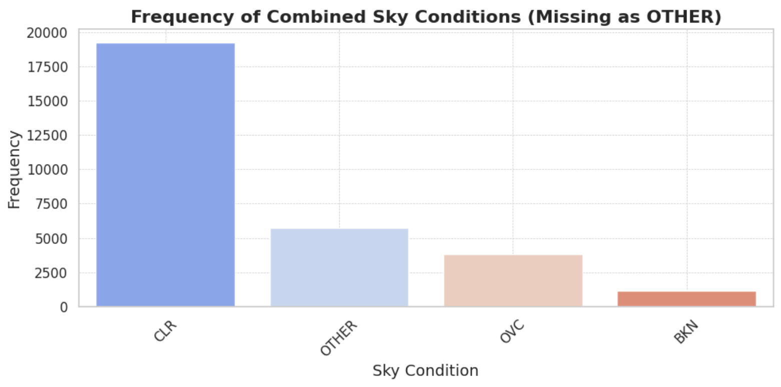


Figure 8: Frequency of Extracted Sky Conditions

The frequency distribution of sky conditions for January 2012. Clear skies (CLR) dominated the month, suggesting generally favorable conditions for aviation. The presence of other conditions such as overcast (OVC) and broken clouds (BKN) were significantly less frequent.

- Temperature

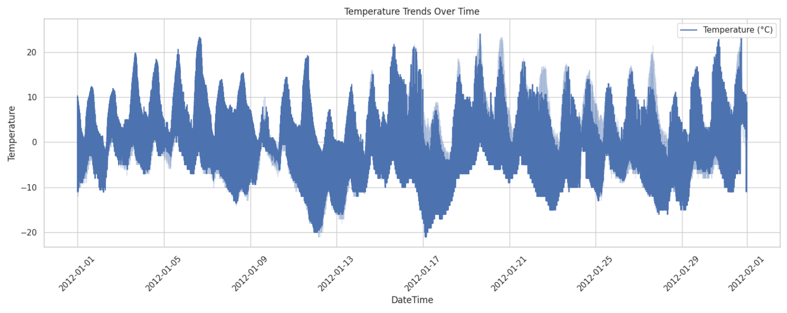


Figure 9: Temperature Trends Over Time

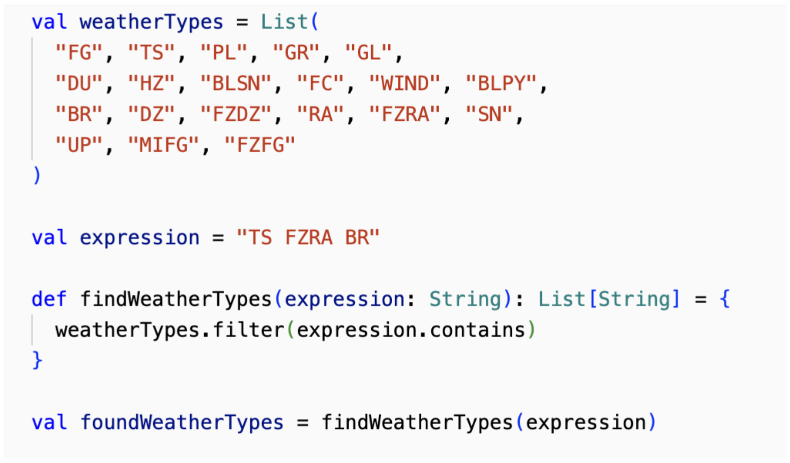
The temperature graph shows significant variability over the course of January 2012, with temperatures ranging widely from well below freezing to above 10 degrees Celsius.

1.2.3 Weather data preprocessing

Challenges and considerations faced during this step:

- Data Completeness: Certain columns have a relatively high number of null entries, suggesting gaps in data collection or occurrence frequency.
- Data Processing: Two challenging columns in this dataset were WeatherType and SkyConditions. These columns posed difficulties because they are composed of multiple values within a single entry, making them relatively complex to work with. The extraction and categorization of data required careful consideration to ensure accuracy and consistency.

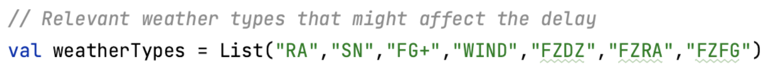
1. **Weather type extraction and selection**
   Weather Types were composed, resulting in numerous combinations (over 200 combinations), despite having only about 20 distinct weather types, as illustrated in (as show in Figure 10). To address this complexity, we developed a method to extract individual weather types from a given expression. Subsequently, we used this method to create columns based on the selected weather types. In these new columns, a value of 0 indicates the absence of the weather type in the expression, while a value of 1 indicates its presence.



```scala
val weatherTypes = List(
  "FG", "TS", "PL", "GR", "GL",
  "DU", "HZ", "BLSN", "FC", "WIND", "BLPY",
  "BR", "DZ", "FZDZ", "RA", "FZRA", "SN",
  "UP", "MIFG", "FZFG"
)

val expression = "TS FZRA BR"

def findWeatherTypes(expression: String): List[String] = {
  weatherTypes.filter(expression.contains)
}

val foundWeatherTypes = findWeatherTypes(expression)
```

Figure 10: Extract Weather Types

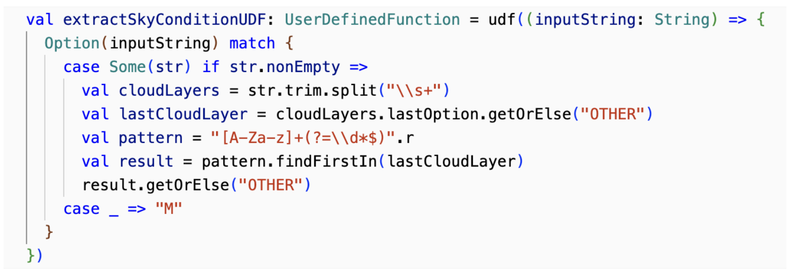
Filtering Relevant Weather Types: we then created a list of weather types (weatherTypes) that are considered relevant for possibly affecting flight delays, including rain (RA), snow (SN), fog (FG+), wind (WIND), and freezing drizzle (FZDZ).



```scala
// Relevant weather types that might affect the delay
val weatherTypes = List("RA","SN","FG+","WIND","FZDZ","FZRA","FZFG")
```

Figure 11: Relevant Weather Types

2. **Sky condition extraction**
   In accordance with the *Local Climatological Data (LCD) Dataset Documentation*, to capture the overall sky condition when up to three cloud layers are reported, we can rely on the description provided for the last layer. In simpler terms, when three layers are mentioned, and the third layer indicates "BKN" (broken clouds), it signifies that the predominant sky condition is "BKN," akin to the definition of "mostly cloudy." We implemented this logic in the following method (Figure 12)

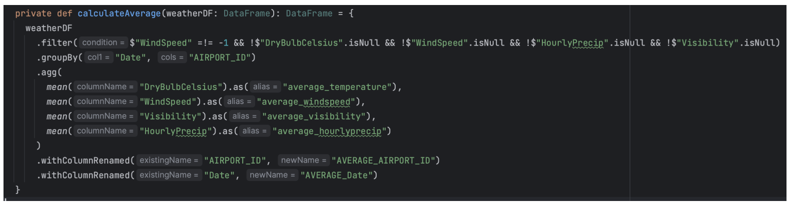


```scala
val extractSkyConditionUDF: UserDefinedFunction = udf((inputString: String) => {
  Option(inputString) match {
    case Some(str) if str.nonEmpty =>
      val cloudLayers = str.trim.split("\\s+")
      val lastCloudLayer = cloudLayers.lastOption.getOrElse("OTHER")
      val pattern = "[A-Za-z]+(?=\\d*$)".r
      val result = pattern.findFirstIn(lastCloudLayer)
      result.getOrElse("OTHER")
    case _ => "M"
  }
})
```

Figure 12: Extract Sky Conditions

3. **Handling null and missing values**
   We implemented distinct approaches to handle missing values, tailoring our methods to the specific characteristics of each column.
   For the "*HourlyPrecip*" column, null data were replaced with 0. This decision aligns with the guidance from the *Local Climatological Data (LCD) Dataset Documentation*, which indicates that no recorded value signifies no precipitation observed or

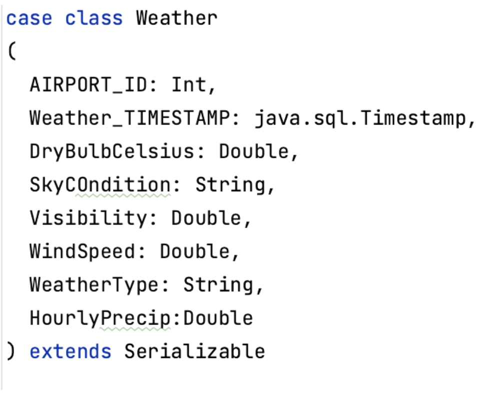
reported for the hour ending at that time. Additionally, a special case was addressed where "T" represented trace amounts of precipitation, and it was also replaced with 0 to denote the absence of precipitation.
The "*WindSpeed*" column underwent a unique treatment. When the value "VR" (variable wind speed) was encountered, we substituted it with -1. This adjustment allowed us to represent wind speed numerically. Subsequently, the column was cast to an integer data type, and missing values were treated as null.
Null values in the "*SkyCondition*" column were previously addressed during the extraction process, as depicted in Figure 12.
Following these initial data cleansing steps, we calculated daily averages for the "*HourlyPrecip*," "*Visibility*," "*WindSpeed*," and "*DryBulbCelsius*" columns, as illustrated in Figure 13. These calculated daily averages served as valuable references for filling missing values in their respective columns, contributing to a more comprehensive and consistent dataset.



```
private def calculateAverage(weatherDF: DataFrame): DataFrame = {
  weatherDF
    .filter( condition = $"WindSpeed" =!= -1 && !$"DryBulbCelsius".isNull && !$"WindSpeed".isNull && !$"HourlyPrecip".isNull && !$"Visibility".isNull)
    .groupBy( col1 = "Date",  cols = "AIRPORT_ID")
    .agg(
      mean( columnName = "DryBulbCelsius").as( alias = "average_temperature"),
      mean( columnName = "WindSpeed").as( alias = "average_windspeed"),
      mean( columnName = "Visibility").as( alias = "average_visibility"),
      mean( columnName = "HourlyPrecip").as( alias = "average_hourlyprecip")
    )
    .withColumnRenamed( existingName = "AIRPORT_ID",  newName = "AVERAGE_AIRPORT_ID")
    .withColumnRenamed( existingName = "Date",  newName = "AVERAGE_Date")
}
```

Figure 13: Calculate Daily Average

4. **Weather Column Selection**
   Weather Case Class Definition: we have defined a Scala case class *Weather* with various attributes like *Airport ID*, *timestamps*, and weather characteristics that we deemed most relevant (*DryBulbCelsius, SkyCondition, Visibility, WindSpeed, WeatherType, HourlyPrecip*).



```
case class Weather
(
  AIRPORT_ID: Int,
  Weather_TIMESTAMP: java.sql.Timestamp,
  DryBulbCelsius: Double,
  SkyCOndition: String,
  Visibility: Double,
  WindSpeed: Double,
  WeatherType: String,
  HourlyPrecip:Double
) extends Serializable
```

Figure 14: Selected Weather Columns

2. Data transformation

In this chapter, we will describe all the steps to transform the Flights and Weather Observations tables into a final table that will include both the features and the target necessary for our machine learning model.
First, we will undertake the task of joining the preprocessed Flights and Weather Observations tables into a unified dataset. We will explore and evaluate two distinct merging methods, comparing them in terms of performance to select the one with the most efficient execution time.
Then, we will proceed to the selection of relevant features for our model which will include information related to both flight and weather data. Additionally, we will engineer the target variable to create a binary classification, distinguishing between delayed and on-time flights according to a certain threshold.
To address the issue of class imbalance within the dataset, we will implement random undersampling. This technique consists of reducing the number of instances in the majority class to achieve a more balanced distribution, ensuring that our model effectively learns from both classes.

2.1. Table join

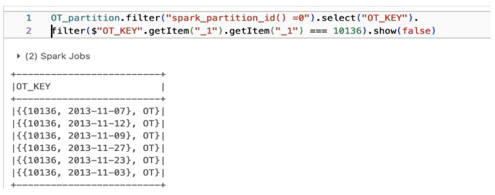
The goal of the join phase is to obtain one table in which each row corresponding to the flight has its relative weather observations: JF = F, Wo , Wd , C, where F contains the flight information, Wo contains the weather at the departure/origin location, Wd contains the weather at the arrival/destination location, C is the delay. It is important to note that the weather observations are:

- for both the departure/origin and arrival/destination locations;
- related to the same hour of scheduled departure and arrival time, and for the eleven hours before, for a total of 12 arrays containing the weather observation for that hour.

2.1.1 Partitioning

To perform the join between the two datasets FT and OT, we partitioned the datasets as described in the article. There were two possible partitioning methods: `hashpartitioner()` and `partitionByRange()`.
Our choice leaned towards the `hashpartitioner` due to its efficiency in our context. For performance enhancement during this stage, we applied the `persist()` method, but limited its usage to the FT dataset only.

Verifying the *hashpartitioner()* distribution
In the partitioning phase, we verified if the data in the FT and OT datasets are distributed on the same partition for the same attached key (`"airport_id"`, `"date"`). This check ensures that the data to be joint is present and will not be "shuffled", a process that is always time-consuming. To show this, we have selected a few examples of joint keys for one dataset (e.g. OT) to ensure that the same joint keys for the other dataset (e.g. FT) are on the same partition, as shown in figures 15 and 16 (on index 0 partition).

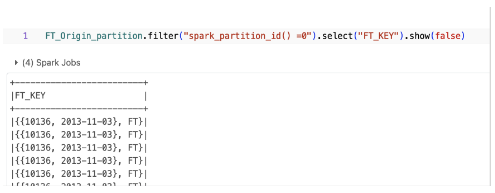


```
OT_partition.filter("spark_partition_id() =0").select("OT_KEY").
filter($"OT_KEY".getItem("_1").getItem("_1") === 10136).show(false)
```

▸ (2) Spark Jobs

```
+------------------------+
|OT_KEY                  |
+------------------------+
|{{10136, 2013-11-07}, OT}|
|{{10136, 2013-11-12}, OT}|
|{{10136, 2013-11-09}, OT}|
|{{10136, 2013-11-27}, OT}|
|{{10136, 2013-11-23}, OT}|
|{{10136, 2013-11-03}, OT}|
+------------------------+
```

Figure 15: Joint key of FT on partition 0



```
FT_Origin_partition.filter("spark_partition_id() =0").select("FT_KEY").show(false)
```

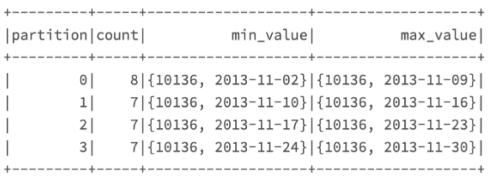
▸ (4) Spark Jobs

```
+------------------------+
|FT_KEY                  |
+------------------------+
|{{10136, 2013-11-03}, FT}|
|{{10136, 2013-11-03}, FT}|
|{{10136, 2013-11-03}, FT}|
|{{10136, 2013-11-03}, FT}|
|{{10136, 2013-11-03}, FT}|
```

Figure 16: Joint key of OT on partition 0

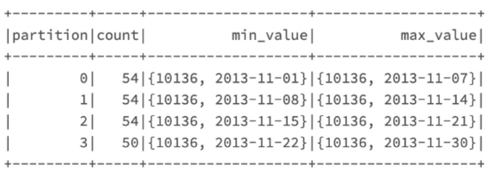
Challenges with using *repartitionByRange()*

RepartitionByRange() cannot work with a partitioning criterion based on two attributes, or it has to be applied to the tuple made up of the two attributes (in this case the joint key (`"airport_id"`, `"date"`). However, on a few samples it became apparent that the distribution of partitions between the two datasets FT and OT was not homogeneous, as shown in the two illustrations below. These two examples show that the intervals between the different partitions of the two datasets are not totally set to the same values.



```
+---------+-----+-------------------+-------------------+
|partition|count|          min_value|          max_value|
+---------+-----+-------------------+-------------------+
|        0|    8|{10136, 2013-11-02}|{10136, 2013-11-09}|
|        1|    7|{10136, 2013-11-10}|{10136, 2013-11-16}|
|        2|    7|{10136, 2013-11-17}|{10136, 2013-11-23}|
|        3|    7|{10136, 2013-11-24}|{10136, 2013-11-30}|
+---------+-----+-------------------+-------------------+
```

Figure 17: Repartition for OT for each partition

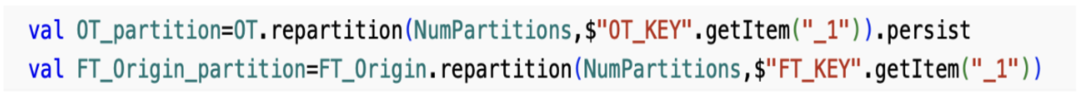


```
+---------+-----+-------------------+-------------------+
|partition|count|          min_value|          max_value|
+---------+-----+-------------------+-------------------+
|        0|   54|{10136, 2013-11-01}|{10136, 2013-11-07}|
|        1|   54|{10136, 2013-11-08}|{10136, 2013-11-14}|
|        2|   54|{10136, 2013-11-15}|{10136, 2013-11-21}|
|        3|   50|{10136, 2013-11-22}|{10136, 2013-11-30}|
+---------+-----+-------------------+-------------------+
```

Figure 18: Repartition for FT for each partition

Optimizing Memory Usage with *persist()* for the OT Dataset

When performing the two successive joins to produce the final dataframe $JF = \{F, Wo, Wd, C\}$, the Weather dataset is used twice. Using the *persist()* method avoids having to reload this dataset twice in memory. On the other hand, it is pointless to apply this method to the FT dataset, whose joined key will be modified between the two stages of the JF table construction process.



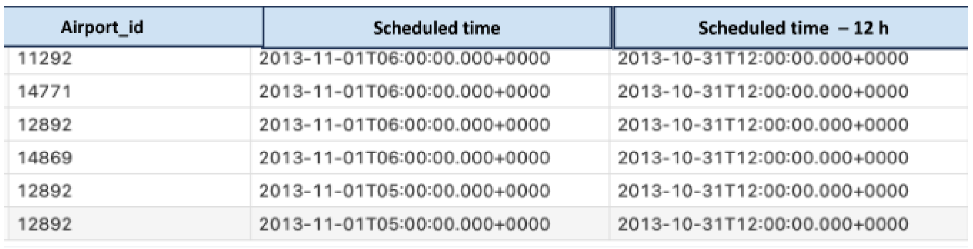
```
val OT_partition=OT.repartition(NumPartitions,$"OT_KEY".getItem("_1")).persist
val FT_Origin_partition=FT_Origin.repartition(NumPartitions,$"FT_KEY".getItem("_1"))
```

Figure 19: Use of persist() method on OT dataset

2.1.2 First join method

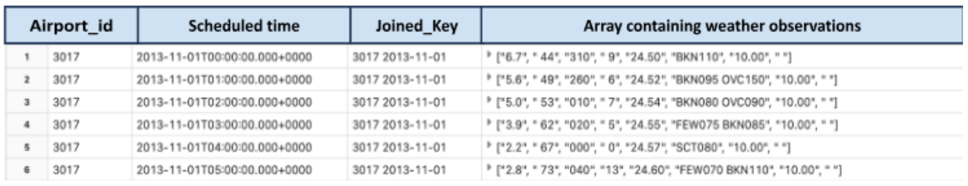
We tested two types of join methods between the FT (for Flights from preprocessing) and OT (for Weahter from preprocessing) tables to obtain the ”JF = F,Wo , Wd , C ” dataframe. The first method is based on a conditional join, the condition being a time range to be respected on the OT data (conditional on the FT schedules). In detail, this method consists of:

1. First, modify the FT table so that each record has a column corresponding to a time range from twelve hours before the scheduled departure (or arrival) time to the scheduled departure (or arrival) time, as shown in the illustration in figure 20.



Airport_id	Scheduled time	Scheduled time – 12 h
11292	2013-11-01T06:00:00.000+0000	2013-10-31T12:00:00.000+0000
14771	2013-11-01T06:00:00.000+0000	2013-10-31T12:00:00.000+0000
12892	2013-11-01T06:00:00.000+0000	2013-10-31T12:00:00.000+0000
14869	2013-11-01T06:00:00.000+0000	2013-10-31T12:00:00.000+0000
12892	2013-11-01T05:00:00.000+0000	2013-10-31T12:00:00.000+0000
12892	2013-11-01T05:00:00.000+0000	2013-10-31T12:00:00.000+0000

Figure 20: Preparing FT table



Airport_id		Scheduled time	Joined_Key	Array containing weather observations
1	3017	2013-11-01T00:00:00.000+0000	3017 2013-11-01	["6.7", " 44", "310", " 9", "24.50", "BKN110", "10.00", " "]
2	3017	2013-11-01T01:00:00.000+0000	3017 2013-11-01	["5.6", " 49", "260", " 6", "24.52", "BKN095 OVC150", "10.00", " "]
3	3017	2013-11-01T02:00:00.000+0000	3017 2013-11-01	["5.0", " 53", "010", " 7", "24.54", "BKN080 OVC090", "10.00", " "]
4	3017	2013-11-01T03:00:00.000+0000	3017 2013-11-01	["3.9", " 62", "020", " 5", "24.55", "FEW075 BKN085", "10.00", " "]
5	3017	2013-11-01T04:00:00.000+0000	3017 2013-11-01	["2.2", " 67", "000", " 0", "24.57", "SCT080", "10.00", " "]
6	3017	2013-11-01T05:00:00.000+0000	3017 2013-11-01	["2.8", " 73", "040", "13", "24.60", "FEW070 BKN110", "10.00", " "]

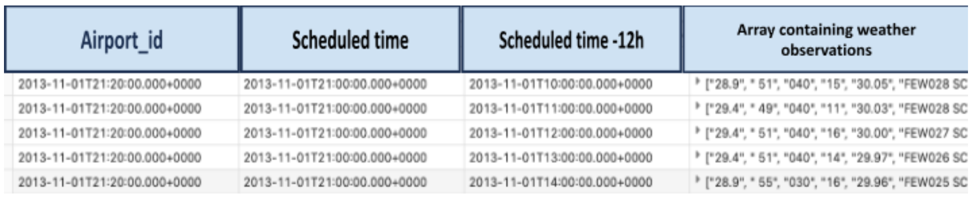
Figure 21: Weather table preparation

2. Then modify the OT table to group all FT attributes in each row in a single tuple as shown in figure 21:

3. After that, a conditional join is performed between the two previous tables, where each FT record (Date, Hour, Airport-id) is associated with an OT record for the same airport and day, provided that the OT time (Hour OT) lies between the flight departure time (Date) and twelve hours before (Date, - 12h). Each FT record is thus duplicated twelve times, each time for an OT record satisfying the criterion (airport-id, date and [Hour-12h, Hour]) as illustrated in figure 22:

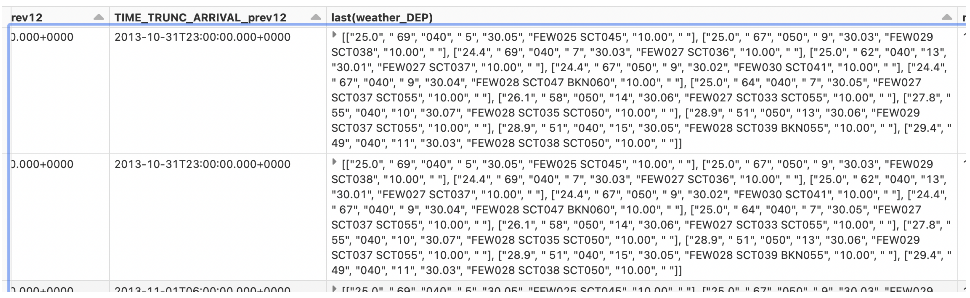
4. As last step, we ordered the weather array - using *window.partitionBy()* combined with *orderBy()* functions- and performed an aggregation by (Date, Hour, Airport-id etc.) using the "collect list" method applied to the tuples containing the weather data in order to group them into a single tuple, as can be seen in figure 23 (and named "last(weatherDEP)").

Note that at the end of this last step, it is important to filter out records with fewer than 12 values in the resulting tuple -here "last(weatherDEP)". This is because the OT data are missing data at certain times, as we saw during the Weather preprocessing phase.
This sequence is executed twice, once on the scheduled departure time and once on the scheduled arrival time.
The code for this join method is presented in the appendix.



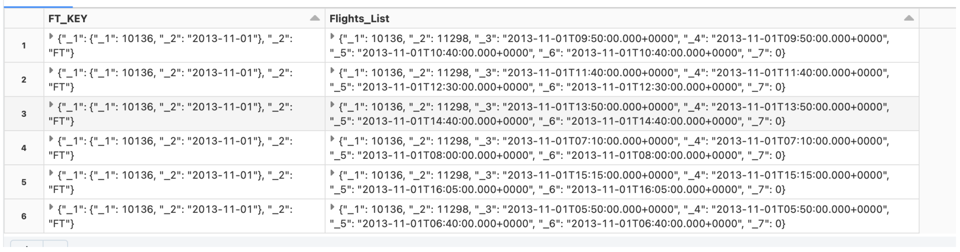
Airport_id	Scheduled time	Scheduled time -12h	Array containing weather observations
2013-11-01T21:20:00.000+0000	2013-11-01T21:00:00.000+0000	2013-11-01T10:00:00.000+0000	["28.9", " 51", "040", "15", "30.05", "FEW028 SC
2013-11-01T21:20:00.000+0000	2013-11-01T21:00:00.000+0000	2013-11-01T11:00:00.000+0000	["29.4", " 49", "040", "11", "30.03", "FEW028 SC
2013-11-01T21:20:00.000+0000	2013-11-01T21:00:00.000+0000	2013-11-01T12:00:00.000+0000	["29.4", " 51", "040", "16", "30.00", "FEW027 SC
2013-11-01T21:20:00.000+0000	2013-11-01T21:00:00.000+0000	2013-11-01T13:00:00.000+0000	["29.4", " 51", "040", "14", "29.97", "FEW026 SC
2013-11-01T21:20:00.000+0000	2013-11-01T21:00:00.000+0000	2013-11-01T14:00:00.000+0000	["28.9", " 55", "030", "16", "29.96", "FEW025 SC

Figure 22: Join between FT and OT on the 12 hours before



rev12	TIME_TRUNC_ARRIVAL_prev12	last(weather_DEP)
).000+0000	2013-10-31T23:00:00.000+0000	[["25.0", " 69", "040", " 5", "30.05", "FEW025 SCT045", "10.00", " "], ["25.0", " 67", "050", " 9", "30.03", "FEW029 SCT038", "10.00", " "], ["24.4", " 69", "040", " 7", "30.03", "FEW027 SCT036", "10.00", " "], ["25.0", " 62", "040", "13", "30.01", "FEW027 SCT037", "10.00", " "], ["24.4", " 67", "050", " 9", "30.02", "FEW030 SCT041", "10.00", " "], ["24.4", " 67", "040", " 9", "30.04", "FEW028 SCT047 BKN060", "10.00", " "], ["25.0", " 64", "040", " 7", "30.05", "FEW027 SCT037 SCT055", "10.00", " "], ["26.1", " 58", "050", "14", "30.06", "FEW027 SCT033 SCT055", "10.00", " "], ["27.8", " 55", "040", "10", "30.07", "FEW028 SCT035 SCT050", "10.00", " "], ["28.9", " 51", "050", "13", "30.06", "FEW029 SCT037 SCT055", "10.00", " "], ["28.9", " 51", "040", "15", "30.05", "FEW028 SCT039 BKN055", "10.00", " "], ["29.4", " 49", "040", "11", "30.03", "FEW028 SCT038 SCT050", "10.00", " "]]
).000+0000	2013-10-31T23:00:00.000+0000	[["25.0", " 69", "040", " 5", "30.05", "FEW025 SCT045", "10.00", " "], ["25.0", " 67", "050", " 9", "30.03", "FEW029 SCT038", "10.00", " "], ["24.4", " 69", "040", " 7", "30.03", "FEW027 SCT036", "10.00", " "], ["25.0", " 62", "040", "13", "30.01", "FEW027 SCT037", "10.00", " "], ["24.4", " 67", "050", " 9", "30.02", "FEW030 SCT041", "10.00", " "], ["24.4", " 67", "040", " 9", "30.04", "FEW028 SCT047 BKN060", "10.00", " "], ["25.0", " 64", "040", " 7", "30.05", "FEW027 SCT037 SCT055", "10.00", " "], ["26.1", " 58", "050", "14", "30.06", "FEW027 SCT033 SCT055", "10.00", " "], ["27.8", " 55", "040", "10", "30.07", "FEW028 SCT035 SCT050", "10.00", " "], ["28.9", " 51", "050", "13", "30.06", "FEW029 SCT037 SCT055", "10.00", " "], ["28.9", " 51", "040", "15", "30.05", "FEW028 SCT039 BKN055", "10.00", " "], ["29.4", " 49", "040", "11", "30.03", "FEW028 SCT038 SCT050", "10.00", " "]]
).000+0000	2013-11-01T06:00:00.000+0000	[["25.0", " 69", "040", " 5", "30.05", "FEW025 SCT045", "10.00", " "], ["25.0", " 67", "050", " 9", "30.03", "FEW029

Figure 23: Grouping the weather arrays in an array containing all of them



	FT_KEY	Flights_List
1	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T09:50:00.000+0000", "_4": "2013-11-01T09:50:00.000+0000", "_5": "2013-11-01T10:40:00.000+0000", "_6": "2013-11-01T10:40:00.000+0000", "_7": 0}
2	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T11:40:00.000+0000", "_4": "2013-11-01T11:40:00.000+0000", "_5": "2013-11-01T12:30:00.000+0000", "_6": "2013-11-01T12:30:00.000+0000", "_7": 0}
3	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T13:50:00.000+0000", "_4": "2013-11-01T13:50:00.000+0000", "_5": "2013-11-01T14:40:00.000+0000", "_6": "2013-11-01T14:40:00.000+0000", "_7": 0}
4	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T07:10:00.000+0000", "_4": "2013-11-01T07:10:00.000+0000", "_5": "2013-11-01T08:00:00.000+0000", "_6": "2013-11-01T08:00:00.000+0000", "_7": 0}
5	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T15:15:00.000+0000", "_4": "2013-11-01T15:15:00.000+0000", "_5": "2013-11-01T16:05:00.000+0000", "_6": "2013-11-01T16:05:00.000+0000", "_7": 0}
6	{"_1": {"_1": 10136, "_2": "2013-11-01"}, "_2": "FT"}	{"_1": 10136, "_2": 11298, "_3": "2013-11-01T05:50:00.000+0000", "_4": "2013-11-01T05:50:00.000+0000", "_5": "2013-11-01T06:40:00.000+0000", "_6": "2013-11-01T06:40:00.000+0000", "_7": 0}

Figure 24: FT data preparation before join -joint key and array with flight information

2.1.3 Second join method

The second method is similar to the one described in the article (Belcastro et al., 2016), as it performs a join between the FT dataset and a table from OT containing, in a single tuple, all 36 weather data records for a given day (i.e. one record per hour, going back to the previous 12 hours of the previous day). This trick of going back to the 12 hours of the previous day covers cases where flight forecast times are at 12 AM.
In detail, this method consists of :

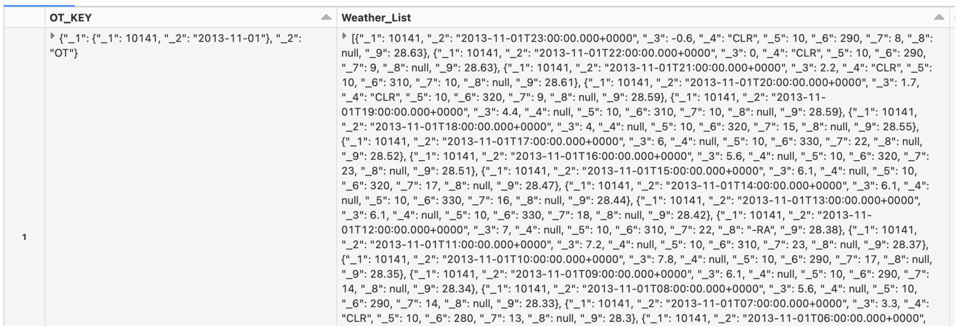
1. First, we create a dataset from the FT dataset, containing one column for the composite key ((airport-id ,date), "FT") and one for the FT dataset (i.e. the 4 selected attributes from Flights), grouped together in a single tuple, as illustrated in figure 24.

2. After that, we created a dataset from the OT dataset, including the composite key ((airport-id,date), "OT") and the weather data for the day (24 hours) and the previous 12 hours, as illustrated in figure 25. Note that at this stage it is important to filter out records with fewer than 36 values (as the Weather table contains some missing data for specific hours).

3. Then, we performed the join between these two tables based on the joint key of each dataset ((airport-id, date) with (airport-id, date)) using the command shown in figure 26.

4. As last step, we selected from the tuple of 36 data from Weather, only those corresponding to the time range between the forecast departure (or arrival) time and twelve hours before, as shown in figure 27.

These steps are executed twice, once on the scheduled departure time and once on the scheduled arrival time.
Note that, although we applied the method described in the article (Belcastro et al.,



```
   OT_KEY                                     Weather_List
1  {"_1": {"_1": 10141, "_2": "2013-11-01"}, "_2":   [{"_1": 10141, "_2": "2013-11-01T23:00:00.000+0000", "_3": -0.6, "_4": "CLR", "_5": 10, "_6": 290, "_7": 8, "_8":
   "OT"}                                            null, "_9": 28.63}, {"_1": 10141, "_2": "2013-11-01T22:00:00.000+0000", "_3": 0, "_4": "CLR", "_5": 10, "_6": 290,
                                                    "_7": 9, "_8": null, "_9": 28.63}, {"_1": 10141, "_2": "2013-11-01T21:00:00.000+0000", "_3": 2.2, "_4": "CLR", "_5":
                                                    10, "_6": 310, "_7": 10, "_8": null, "_9": 28.61}, {"_1": 10141, "_2": "2013-11-01T20:00:00.000+0000", "_3": 1.7,
                                                    "_4": "CLR", "_5": 10, "_6": 320, "_7": 9, "_8": null, "_9": 28.59}, {"_1": 10141, "_2": "2013-11-
                                                    01T19:00:00.000+0000", "_3": 4.4, "_4": null, "_5": 10, "_6": 310, "_7": 10, "_8": null, "_9": 28.59}, {"_1": 10141,
                                                    "_2": "2013-11-01T18:00:00.000+0000", "_3": 4, "_4": null, "_5": 10, "_6": 320, "_7": 15, "_8": null, "_9": 28.55},
                                                    {"_1": 10141, "_2": "2013-11-01T17:00:00.000+0000", "_3": 6, "_4": null, "_5": 10, "_6": 330, "_7": 22, "_8": null,
                                                    "_9": 28.52}, {"_1": 10141, "_2": "2013-11-01T16:00:00.000+0000", "_3": 5.6, "_4": null, "_5": 10, "_6": 320, "_7":
                                                    23, "_8": null, "_9": 28.51}, {"_1": 10141, "_2": "2013-11-01T15:00:00.000+0000", "_3": 6.1, "_4": null, "_5": 10,
                                                    "_6": 320, "_7": 17, "_8": null, "_9": 28.47}, {"_1": 10141, "_2": "2013-11-01T14:00:00.000+0000", "_3": 6.1, "_4":
                                                    null, "_5": 10, "_6": 330, "_7": 16, "_8": null, "_9": 28.44}, {"_1": 10141, "_2": "2013-11-01T13:00:00.000+0000",
                                                    "_3": 6.1, "_4": null, "_5": 10, "_6": 330, "_7": 18, "_8": null, "_9": 28.42}, {"_1": 10141, "_2": "2013-11-
                                                    01T12:00:00.000+0000", "_3": 7, "_4": null, "_5": 10, "_6": 310, "_7": 22, "_8": "-RA", "_9": 28.38}, {"_1": 10141,
                                                    "_2": "2013-11-01T11:00:00.000+0000", "_3": 7.2, "_4": null, "_5": 10, "_6": 310, "_7": 23, "_8": null, "_9": 28.37},
                                                    {"_1": 10141, "_2": "2013-11-01T10:00:00.000+0000", "_3": 7.8, "_4": null, "_5": 10, "_6": 290, "_7": 17, "_8": null,
                                                    "_9": 28.35}, {"_1": 10141, "_2": "2013-11-01T09:00:00.000+0000", "_3": 6.1, "_4": null, "_5": 10, "_6": 290, "_7":
                                                    14, "_8": null, "_9": 28.34}, {"_1": 10141, "_2": "2013-11-01T08:00:00.000+0000", "_3": 5.6, "_4": null, "_5": 10,
                                                    "_6": 290, "_7": 14, "_8": null, "_9": 28.33}, {"_1": 10141, "_2": "2013-11-01T07:00:00.000+0000", "_3": 3.3, "_4":
                                                    "CLR", "_5": 10, "_6": 280, "_7": 13, "_8": null, "_9": 28.3}, {"_1": 10141, "_2": "2013-11-01T06:00:00.000+0000",
```

Figure 25: OT data preparation before join - joint key and array with flight information

```scala
val FOT_joined_Origin= OT_partition.join(FT_Origin_partition,OT_partition("OT_KEY").getItem("_1") ===
FT_Origin_partition("FT_KEY").getItem("_1"),"inner")
```

Figure 26: Join based on joint keys FT and OT

2016) by creating a composite key (joined key, table name), we were unable to show the benefit of having the table name included in the key.

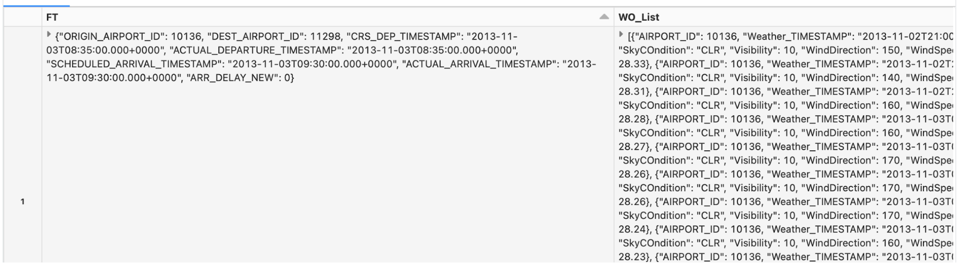
2.1.4 Performance comparison of the two join methods

In terms of performance, the second method is faster, the main reason being that :

- With the *first method,* during the join preparation phase, each Flights record is duplicated twelve times (1 time per Weather data item per hour, from the forecast departure (or arrival) time up to twelve hours before),

- whereas the *second method* involves the creation of an OT table corresponding roughly to one and a half times the size of the initial Weather table (for each day, 36 records are grouped together, corresponding to division by 24 followed by multiplication by 36).

We ran several tests to check the execution time of these two methods. The measured time is taken from the transformation of the tables to the execution of a "spark action" (in this case, we chose a *show()*).
We also considered a third method similar to the conditional join (method 1) which consists of using the "withColumn(Columname, lag(col(Columnname),.over(windowSpec))" method, which creates an intermediate table duplicating each Flights record 12 times.

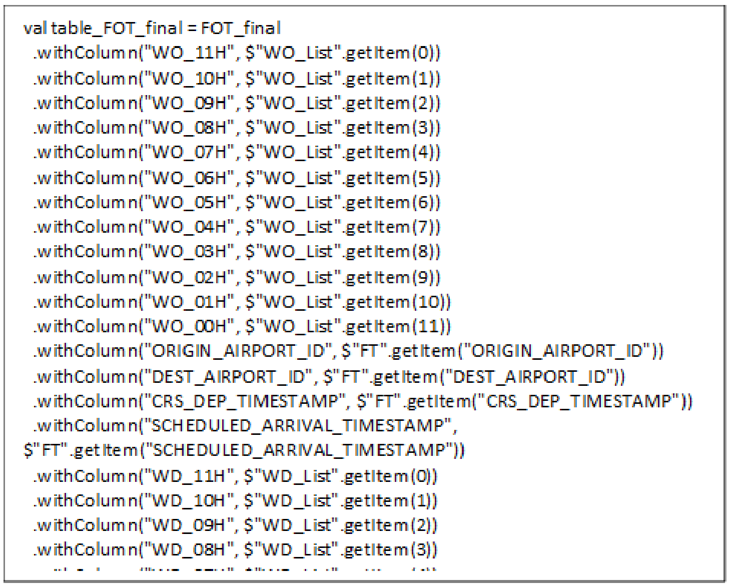


```
   FT                                                                                      WO_List
1  {"ORIGIN_AIRPORT_ID": 10136, "DEST_AIRPORT_ID": 11298, "CRS_DEP_TIMESTAMP": "2013-11-      [{"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-02T21:00
   03T08:35:00.000+0000", "ACTUAL_DEPARTURE_TIMESTAMP": "2013-11-03T08:35:00.000+0000",      "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 150, "WindSpe
   "SCHEDULED_ARRIVAL_TIMESTAMP": "2013-11-03T09:30:00.000+0000", "ACTUAL_ARRIVAL_TIMESTAMP": "2013-   28.33}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-02T
   11-03T09:30:00.000+0000", "ARR_DELAY_NEW": 0}                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 140, "WindSpe
                                                                                            28.31}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-02T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 160, "WindSpe
                                                                                            28.28}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 160, "WindSpe
                                                                                            28.27}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 170, "WindSpe
                                                                                            28.26}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 170, "WindSpe
                                                                                            28.26}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 170, "WindSpe
                                                                                            28.24}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
                                                                                            "SkyCOndition": "CLR", "Visibility": 10, "WindDirection": 160, "WindSpe
                                                                                            28.23}, {"AIRPORT_ID": 10136, "Weather_TIMESTAMP": "2013-11-03T
```

Figure 27: Dataset obtained from the 1st join on departure data

Numbr of tries	Join method 1	Join method 2
1	483 s	268 s (630 on LAMSADE)
2	495 s	271 s
3	579 s	270 s
4	528 s	266 s
5	526 s	277 s

Table 1: Join methods performance comparison



```
val table_FOT_final = FOT_final
  .withColumn("WO_11H", $"WO_List".getItem(0))
  .withColumn("WO_10H", $"WO_List".getItem(1))
  .withColumn("WO_09H", $"WO_List".getItem(2))
  .withColumn("WO_08H", $"WO_List".getItem(3))
  .withColumn("WO_07H", $"WO_List".getItem(4))
  .withColumn("WO_06H", $"WO_List".getItem(5))
  .withColumn("WO_05H", $"WO_List".getItem(6))
  .withColumn("WO_04H", $"WO_List".getItem(7))
  .withColumn("WO_03H", $"WO_List".getItem(8))
  .withColumn("WO_02H", $"WO_List".getItem(9))
  .withColumn("WO_01H", $"WO_List".getItem(10))
  .withColumn("WO_00H", $"WO_List".getItem(11))
  .withColumn("ORIGIN_AIRPORT_ID", $"FT".getItem("ORIGIN_AIRPORT_ID"))
  .withColumn("DEST_AIRPORT_ID", $"FT".getItem("DEST_AIRPORT_ID"))
  .withColumn("CRS_DEP_TIMESTAMP", $"FT".getItem("CRS_DEP_TIMESTAMP"))
  .withColumn("SCHEDULED_ARRIVAL_TIMESTAMP",
$"FT".getItem("SCHEDULED_ARRIVAL_TIMESTAMP"))
  .withColumn("WD_11H", $"WD_List".getItem(0))
  .withColumn("WD_10H", $"WD_List".getItem(1))
  .withColumn("WD_09H", $"WD_List".getItem(2))
  .withColumn("WD_08H", $"WD_List".getItem(3))
```

Figure 28: Saving the joint table

Despite this, we decided to focus and compare only two join methods as previously explained.

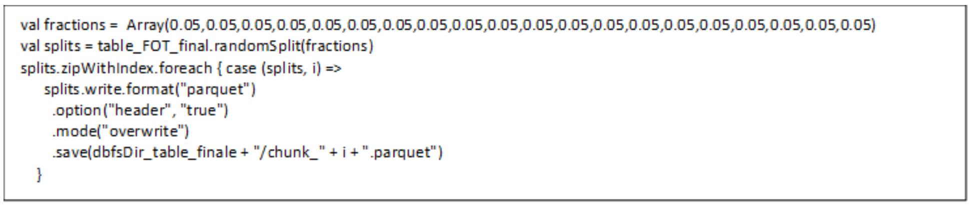
2.1.5 Final joint table

It appeared that saving the joint table in .csv format was not compatible with a dataset containing tuples (F, Wo , Wd) where F is in the form of (Airport-Id, ...) and W in the form of (Airport-id, Timestamp, DryBulbCelsius ...). Rather than transforming the FT dataset to have as many columns as values in the various tuples, we have chosen to save a dataset whose structure corresponds to:

- one column per time slot of the Wo (weather at origin location) and Wd (weather at destination location) data,
- one column for the flight information data,
- one column for the delay information (the model target).

Figure 28 shows the code used to create the final structure of the table to be saved.

We implement this option in Parquet format. This appeared to be particularly costly in terms of memory resources, and the final FT dataset had to be split into 20 ”chunks” to avoid ”out of memory” messages, as shown in figure 29.



```
val fractions =  Array(0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05)
val splits = table_FOT_final.randomSplit(fractions)
splits.zipWithIndex.foreach { case (splits, i) =>
    splits.write.format("parquet")
      .option("header", "true")
      .mode("overwrite")
      .save(dbfsDir_table_finale + "/chunk_" + i + ".parquet")
  }
```

Figure 29: Dataset split in 20 chunks in parquet format

2.2. Model features and target

After having joined the two tables, we selected the features that would be used as input of the model, both about flight information and weather observations.

Flight information for both departure and arrival includes:

- *Hours of scheduled departure/arrival* - numerical variable.
- *Airport of departure/arrival* - categorical variable. Even if the ID are integers, this does not make sense in the model. Therefore we treated this variable as categorical, transforming it to string (cast(”String”)) and treating it in the pipeline via a ”stringindexer” and a ”maxcat” based on the number of airports (270).
- *Month of departure/arrival* - numerical variable. This variable was not mentioned in the paper that we are using as a reference for this project, but we decided to include it as it may indicate some seasonality and relation with weather conditions, and therefore delays. We did not include it in our model but we kept it for future use.

Weather information includes in an array the observations for each hour, from the same hour of scheduled departure/arrival back to 12 hours before. Each array includes the following variables:

- *temperature* - numerical variable.
- *wind speed* - numerical variable.
- *sky condition* - categorical variable
- *visibility* - numerical variable
- *weather type* - categorical variable.
- *hourly precipitation* - numerical variable.

The numeric input features have not been normalized since this step is not necessary when applying a Random Forest model. This is because of several reasons. In a Random Forest, the tree structure is determined independently for each feature, and the scale of features does not impact the decision boundaries as split points are based on the relative ordering of feature values. Unlike distance-based algorithms, Random Forest does not rely on distance metrics, so the scale of features doesn’t significantly affect the outcome. This eliminates the need for normalization to ensure equal feature contributions. Moreover, Random Forests are robust to outliers as they divide data into bins based on feature values reducing their impact on the overall tree structure.

In order to engineer the target of our model, we have considered the ’arrival delay’ column, which indicates the delay in minutes of each flight. This column has been transformed from an integer into a binary value which is 1 when the flight is delayed and 0 when the flight is on time. This categorization is based on a threshold of minutes, which we made it vary in order to check the different results of the model.

2.3. Data sampling

The dataset we have prepared for our model is characterized by a strong imbalance between the two classes: the number of on-time flights is significantly higher than the number of delayed flights. This imbalance varies with the delay threshold, and in particular increases as the delay threshold decreases. For example, the ratio is 23.79 percent (minclass.count= 620 269/maxclass.count= 2 606 350) between the minority class (late) and the majority class (on time) for a delay threshold set at 15 min. The ratio is 6.3 percent for a delay threshold of 60 min.
Unbalanced datasets in machine learning present inherent challenges, as models naturally favor the majority class in order to minimize overall errors. This bias results in greater accuracy for the majority class, while often causing poor classification of the minority class. In addition, the scarcity of examples of the minority class negatively impacts the model ability to learn and understand the unique patterns associated with that class.
To solve the problem of class imbalance within the dataset, we used the random subsampling method, which consists of randomly deleting observations from the majority class.

2.4. Data pipeline

In order to handle the data that would feed our model, a Spark pipeline has been used. The pipeline incorporates the *StringIndexer* transformation as part of the encoding process for categorical features. By default, this is ordered by label frequencies, so the most frequent label gets index 0. This transformation assigns a unique numerical index to each distinct category, facilitating the Random Forest algorithm’s ability to work with categorical data. This has been applied to all categorical variables and to flight IDs.
Another parameter that has been adjusted in the pipeline is *maxCat*. It determines the maximum number of categories allowed for categorical features (for example, there are 270 different airports). This is a crucial consideration as it impacts how Spark processes categorical data within the Random Forest algorithm. Setting an appropriate maxCat is fundamental to balancing computational efficiency and model performance.
Additionally, for handling invalid entries in the data, we utilize the handleInvalid option (on StringIndexer, VectorAssembler and VectorIndexer of the pipelines). The choice of how to handle invalid entries can significantly affect the quality and completeness of the training dataset. This parameter can take several values, including ”skip” -to ignore the rows that contain invalid values- and ’keep’ (we tested both in our models) .

3. Model and evaluation

The article Using Scalable Data Mining for Predicting Flight Delays (Belcastro et al., 2016) reported that the Random Forest model was identified as the one that gave the

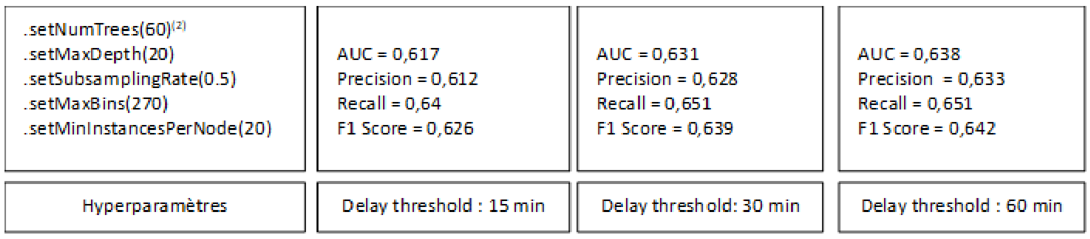
best results for predicting flight delays. In our project, we therefore opted for its implementation.
Random Forest is an ensemble machine learning method that in our project we have used for a binary classification task. It consists of several decision trees that collectively make predictions: each decision tree in the forest casts a vote for the class, and the majority vote determines the final prediction. The Random Forest model can have several advantages such as its robustness, accuracy and its resistance to over-fitting.
When building our model we considered several hyperparameters for its tuning:

- Number of Trees, which determines the number of decision trees in the forest. A higher number of trees generally leads to better performance, but it also increases computation time.
- Tree Depth, which determines the maximum depth of each decision tree. Controlling tree depth helps prevent overfitting.
- Minimum Samples per Leaf: which determines the minimum number of samples required to be in a leaf node. This parameter helps control tree depth and overfitting.
- Maximum Features, the maximum number of features considered for each split.

The Random Forest model has been applied on the train set and then evaluated on the test set. Being in the context of a binary classification, we can consider several metrics:

- Accuracy, which measures the overall correctness of a classification model. It calculates the ratio of correctly predicted instances (both true positives and true negatives) to the total number of instances in the dataset.
- Recall, which measures the ability of a classification model to correctly identify all relevant instances within a particular class. A high recall indicates that the model is good at identifying most of the positive instances, minimizing false negatives.
- Precision, which measures the ability of a classification model to make positive predictions that are actually correct. High precision indicates that when the model predicts positive, it is more likely to be correct, reducing false positives.
- F1 score, which is the harmonic mean of precision and recall. High F1 score indicates both high precision and high recall.

In this project, we will assess various models using the F1 score as our evaluation metric. The F1 score is chosen for its ability to balance between precision and recall, making it particularly suitable for scenarios where the dataset is imbalanced. The accuracy, on the other hand, is not suited for our case and we will not consider it.
In the following paragraphs we will provide results for our baseline model, based on only Flight information, and more complex models which include the weather observations at scheduled departure and arrival time. The grid search technique will be used for Hyperparameters tuning and F1 score for evaluation.



.setNumTrees(60)[2]
.setMaxDepth(20)
.setSubsamplingRate(0.5)
.setMaxBins(270)
.setMinInstancesPerNode(20)
	AUC = 0,617
	Precision = 0,612
	Recall = 0,64
	F1 Score = 0,626
		AUC = 0,631
		Precision = 0,628
		Recall = 0,651
		F1 Score = 0,639
			AUC = 0,638
			Precision = 0,633
			Recall = 0,651
			F1 Score = 0,642

Hyperparamètres	Delay threshold : 15 min	Delay threshold: 30 min	Delay threshold : 60 min

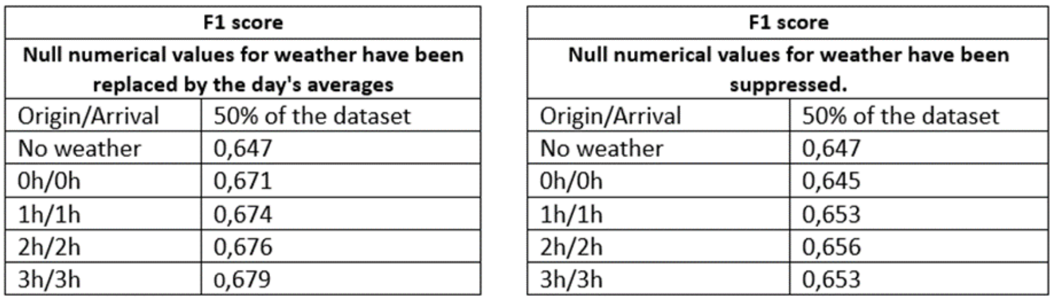
Figure 30: Baseline model results

3.1. Baseline model

Following the methodology described in the article (Belcastro et al., 2016), we first created a baseline model which included only the flight information of both departure and arrival. This enabled us to evaluate more complex models that also include weather information. As a preliminary step, we searched for the best hyperparameters (via a gridsearch) on a quarter of the dataset, the values of which are presented in the table on the left. The accuracy obtained is 63.8 percent (versus 69.1 percent in the article) for a delay threshold of 60 minutes (figure 30)
At this stage, the biggest difficulty was avoiding the "OutOfMemoryError" error message, whether for gridsearch or model training. For gridsearch, including with only a quarter of the dataset, we proceeded parameter by parameter. We limited ourselves to this value, as we found that large values (e.g. 50 or even 70) systematically led to "OutOfMemoryError" errors.
Out of curiosity, we tested a 'baseline model' in which the actual departure and arrival dates/times were retained, in order to estimate their links with the target. For a threshold set at 60 min, we obtained an accuracy of 0.95 percent ('Precision' of 0.91 and 'Recall' of 0.98); this naturally indicates a high degree of adherence between the attributes 'actual departure timestamp' and 'actual arrival departure', and a corruption of the prediction model by a 'linkage' between some of the attributes and the target.

3.2. Model results comparing different pre-processing approaches on Weather table

In this section we look at the impact of replacing the missing numerical values in Weather with the day's averages, compared to a model where the null values are suppressed (added on Appendix 6.2). Considering the F1 score as evaluation metric, the first approach registered better results as shown in figure 31.
To note that we used 50 percent of the dataset and the following hyperparameters:
.setNumTree(60)
.setMaxDepth(20)
.setSubsamplingRate(0.5)
.setMaxBins(270)
.setMinInstancesPerNode(20)
Et handleInvalid = keep



F1 score	
Null numerical values for weather have been replaced by the day's averages	
Origin/Arrival	50% of the dataset
No weather	0,647
0h/0h	0,671
1h/1h	0,674
2h/2h	0,676
3h/3h	0,679

F1 score	
Null numerical values for weather have been suppressed.	
Origin/Arrival	50% of the dataset
No weather	0,647
0h/0h	0,645
1h/1h	0,653
2h/2h	0,656
3h/3h	0,653

Figure 31: Model results comparing different pre-processing approaches on Weather table

3.3. Model results comparing different numbers of hours of weather observations

The model has been tested considering the weather observations over various time spans. The hyperparameters are the same to the ones used on the previous paragraph.

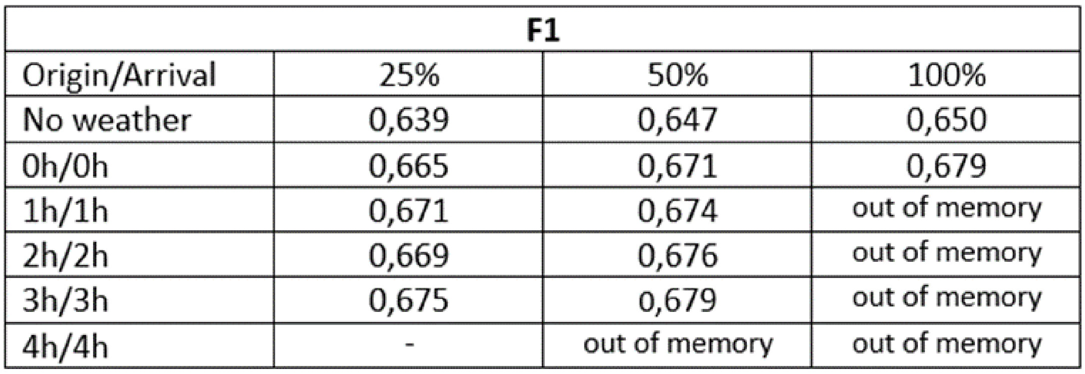
Regarding the findings, it is crucial to acknowledge that the presented values serve as general indicators, and variations may arise in successive tests, likely associated with the randomness introduced during the dataset split for training and testing.

Upon examination, a clear trend emerges, indicating that enlarging the dataset contributes positively to the model performance and improvement of the F1 score (figure 32). Additionally, is worth mentioning that predictions incorporating only the weather data corresponding to the forecast times record slightly better results than predictions made without weather data. Furthermore, extending the time range have a positive influence on the F1 score. However, the improvements associated with broader time slots are not prominently evident or, in some cases, weak. This can be attributed to several factors such as:

- the relatively limited size of the dataset
- constraints imposed by the computation power of the cluster, which limited the exploration of optimal hyperparameters and constrained weather time slots to 2 hours with forecast departures/arrivals.
- the impact of choices made concerning weather attributes, including the processing of WeatherType and SkyCondition attribute values, contributes to the observed outcomes.

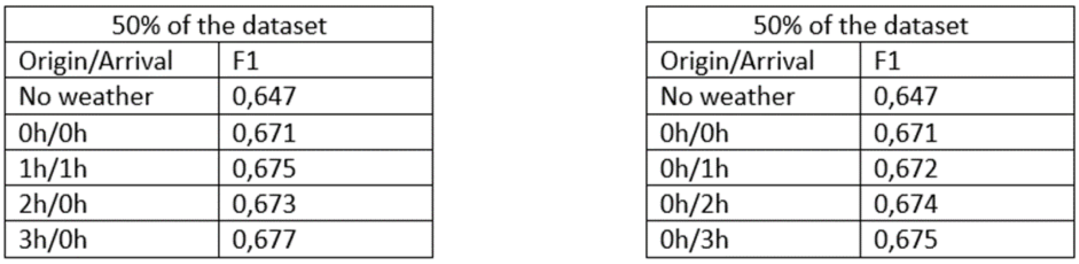
Regarding the relative significance of weather data between the departure and arrival airports, the conducted tests suggest an equal contribution of departure airport weather data to the forecast quality of delays in our dataset (figure 33).

For an additional evaluation, we conducted a series of tests, modifying the "HandleInvalid" parameter from "keep" to "skip." The findings suggest that removing records containing at least one Null value contributes to enhancements in the model's performance (figure 34).



F1			
Origin/Arrival	25%	50%	100%
No weather	0,639	0,647	0,650
0h/0h	0,665	0,671	0,679
1h/1h	0,671	0,674	out of memory
2h/2h	0,669	0,676	out of memory
3h/3h	0,675	0,679	out of memory
4h/4h	-	out of memory	out of memory

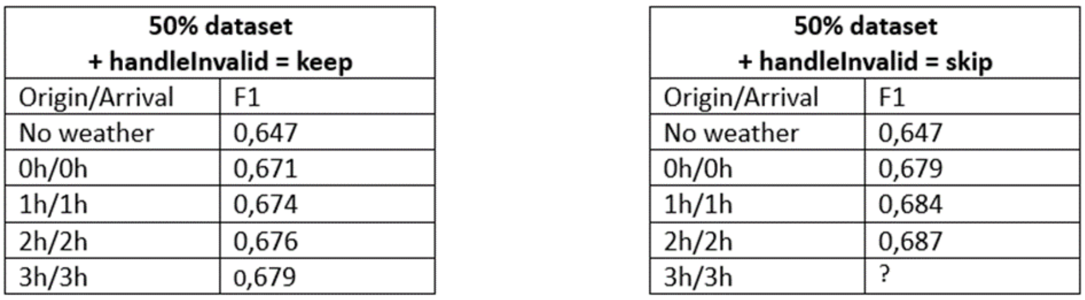
Figure 32: Model performance obtained with a delay threshold set at 60 min



50% of the dataset	
Origin/Arrival	F1
No weather	0,647
0h/0h	0,671
1h/1h	0,675
2h/0h	0,673
3h/0h	0,677

50% of the dataset	
Origin/Arrival	F1
No weather	0,647
0h/0h	0,671
0h/1h	0,672
0h/2h	0,674
0h/3h	0,675

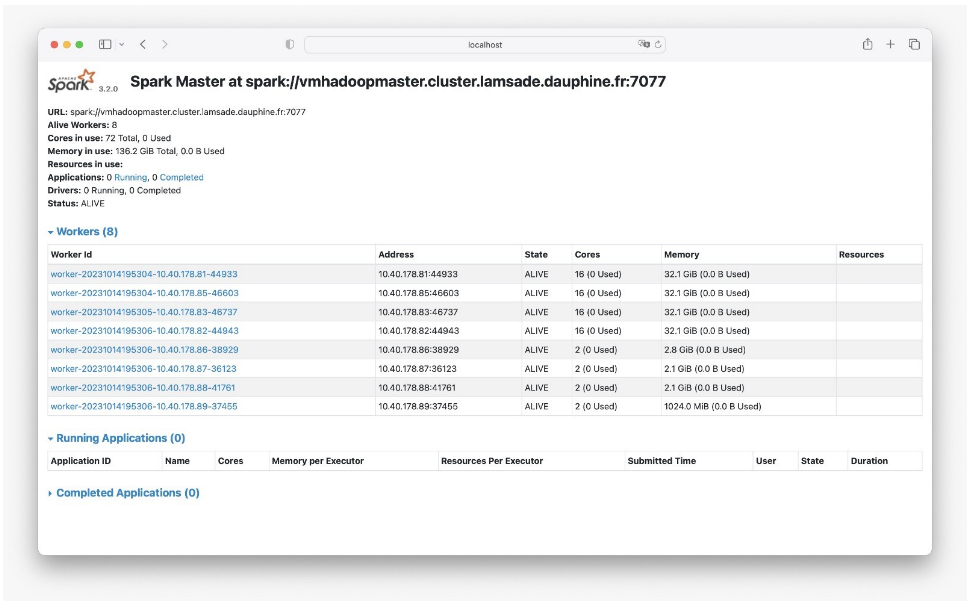
Figure 33: Model performance comparison with different number of hours considered at departure and arrival



50% dataset + handleInvalid = keep	
Origin/Arrival	F1
No weather	0,647
0h/0h	0,671
1h/1h	0,674
2h/2h	0,676
3h/3h	0,679

50% dataset + handleInvalid = skip	
Origin/Arrival	F1
No weather	0,647
0h/0h	0,679
1h/1h	0,684
2h/2h	0,687
3h/3h	?

Figure 34: Model performance comparison with different treatments of invalid weather observations


localhost

Spark 3.2.0 **Spark Master at spark://vmhadoopmaster.cluster.lamsade.dauphine.fr:7077**

URL: spark://vmhadoopmaster.cluster.lamsade.dauphine.fr:7077
Alive Workers: 8
Cores in use: 72 Total, 0 Used
Memory in use: 136.2 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (8)

Worker Id	Address	State	Cores	Memory	Resources
worker-20231014195304-10.40.178.81-44933	10.40.178.81:44933	ALIVE	16 (0 Used)	32.1 GiB (0.0 B Used)	
worker-20231014195304-10.40.178.85-46603	10.40.178.85:46603	ALIVE	16 (0 Used)	32.1 GiB (0.0 B Used)	
worker-20231014195305-10.40.178.83-46737	10.40.178.83:46737	ALIVE	16 (0 Used)	32.1 GiB (0.0 B Used)	
worker-20231014195306-10.40.178.82-44943	10.40.178.82:44943	ALIVE	16 (0 Used)	32.1 GiB (0.0 B Used)	
worker-20231014195306-10.40.178.86-38929	10.40.178.86:38929	ALIVE	2 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20231014195306-10.40.178.87-36123	10.40.178.87:36123	ALIVE	2 (0 Used)	2.1 GiB (0.0 B Used)	
worker-20231014195306-10.40.178.88-41761	10.40.178.88:41761	ALIVE	2 (0 Used)	2.1 GiB (0.0 B Used)	
worker-20231014195306-10.40.178.89-37455	10.40.178.89:37455	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

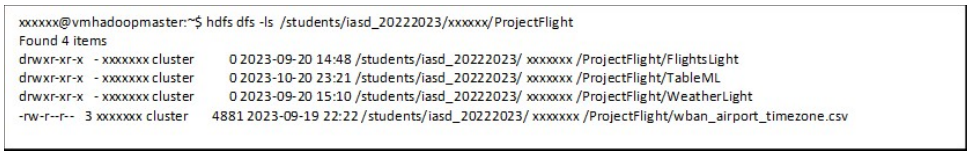
Figure 35: LAMSADE Cluster specification

4. Application Development and Execution

Our application has been written in Scala, Apache Spark framework and its ML libraries, using IntelliJ IDEA as integrated development environment (IDE). Then, the program has been executed on the Lamsade cluster.
Apache Spark allowed us to work with large-scale data thanks to its distributed computing system framework designed for big data processing and analytics. We also exploit MLlib which is a core component of Spark that offers a variety of machine learning algorithms for classification, regression, clustering and more.
The choice of writing our program in scala offered several advantages. The most important one is related to the fact that Scala is a "typed" language: errors are raised at compile time (instead of runtime as in Python), leading to more reliable code. Other advantages include its performance -Scala is often faster than Python for certain data processing tasks- and the fact of being a functional programming language which is in line with the way Spark's core APIs are designed.
The amount of data treated in our project required the use of a cluster, a group of interconnected computers or servers that work together as a single, unified system. We used the LAMSADE cluster that has the characteristics shown in figure 35.
In the next paragraphs we will provide details about our program structure, installation and execution.

4.1. Project Organization and Data Storage Structure

The project has been organized into two distinct applications: the first one, referred to as **dataprocessingapp_2.12-0.1.jar**, is primarily responsible for data preprocessing and transformation. The second application, **delaypredictionapp_2.12-0.1.jar**, focuses on machine learning tasks.
Data from the first phase are stored in parquet format, which stores data in a columnar format. The advantage of creating two applications, saving the preprocessed table, is re-



```
xxxxxx@vmhadoopmaster:~$ hdfs dfs -ls  /students/iasd_20222023/xxxxxx/ProjectFlight
Found 4 items
drwxr-xr-x   - xxxxxxx cluster          0 2023-09-20 14:48 /students/iasd_20222023/ xxxxxxx /ProjectFlight/FlightsLight
drwxr-xr-x   - xxxxxxx cluster          0 2023-10-20 23:21 /students/iasd_20222023/ xxxxxxx /ProjectFlight/TableML
drwxr-xr-x   - xxxxxxx cluster          0 2023-09-20 15:10 /students/iasd_20222023/ xxxxxxx /ProjectFlight/WeatherLight
-rw-r--r--   3 xxxxxxx cluster       4881 2023-09-19 22:22 /students/iasd_20222023/ xxxxxxx /ProjectFlight/wban_airport_timezone.csv
```

Figure 36: Structure for data storage

lated to the possibility to run the model several times with different parameters without executing again the preprocessing part.
Both applications refer to the same storage tree to retrieve data (Flights, Weather Time-Zone and data from the preprocessing phase) which follows the following structure as shown on figure 36:

- Project root with various data sub-directories: "./ProjectFlight".
- Flights data subdirectory: "./ProjectFlight/FlightsLight".
- Weather data subdirectory: "./ProjectFlight/WeatherLight
- Location of timezone file: "./ProjectFlight/wban-airport-timezone.csv".
- A user-selected subdirectory housing pre-processed data.

4.2. Run the first application: data-processing

The first application contained in **dataprocessingapp_2.12-0.1.jar** executes the processing and transformation steps which allow cleaning the Flights and Weather Observations tables and at the end joining them. The output of this application is saved in the directory chosen by the user in Parquet format.

The spark-submit command is:

Listing 1: Spark-submit data-process

```
spark-submit \
    --executor-cores 12 \
    --num-executors 4 \
    --executor-memory 8G \
    --class com.DataProcessingApplication \
    --master local \
    dataprocessingapp_2.12-0.1.jar
```

While running the data-processing, the user will be given instructions on how to get the input data or how to store them (figure 37).

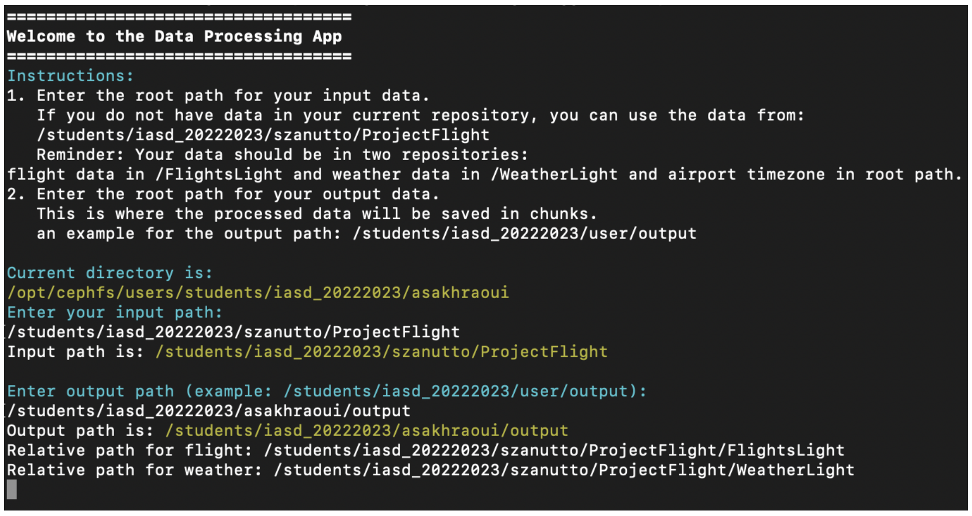

```
==================================
Welcome to the Data Processing App
==================================
Instructions:
1. Enter the root path for your input data.
   If you do not have data in your current repository, you can use the data from:
   /students/iasd_20222023/szanutto/ProjectFlight
   Reminder: Your data should be in two repositories:
flight data in /FlightsLight and weather data in /WeatherLight and airport timezone in root path.
2. Enter the root path for your output data.
   This is where the processed data will be saved in chunks.
   an example for the output path: /students/iasd_20222023/user/output

Current directory is:
/opt/cephfs/users/students/iasd_20222023/asakhraoui
Enter your input path:
/students/iasd_20222023/szanutto/ProjectFlight
Input path is: /students/iasd_20222023/szanutto/ProjectFlight

Enter output path (example: /students/iasd_20222023/user/output):
/students/iasd_20222023/asakhraoui/output
Output path is: /students/iasd_20222023/asakhraoui/output
Relative path for flight: /students/iasd_20222023/szanutto/ProjectFlight/FlightsLight
Relative path for weather: /students/iasd_20222023/szanutto/ProjectFlight/WeatherLight
```

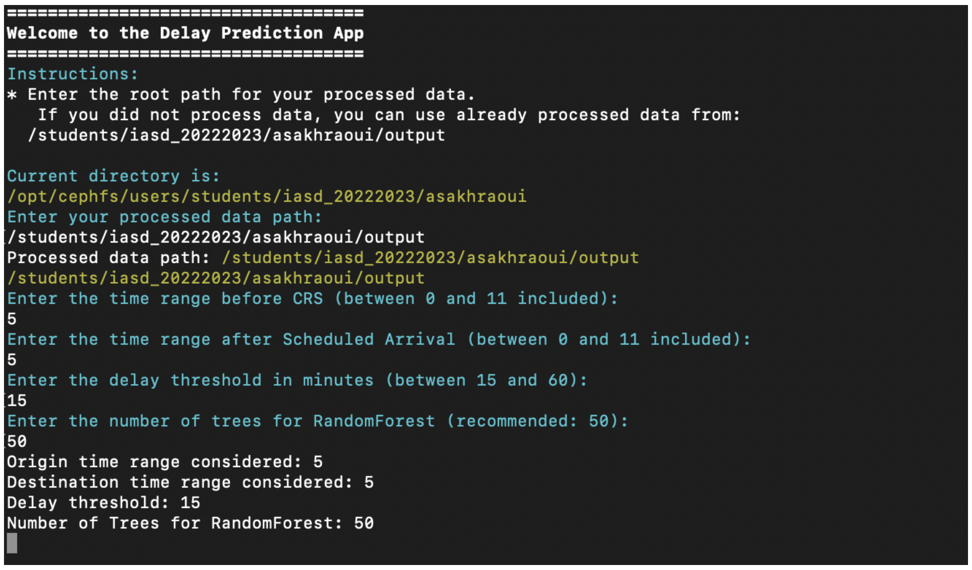
Figure 37: Input required by the program - data-process

4.3. Run the second application: delay-prediction

The second application **delaypredictionapp_2.12-0.1.jar** takes the output of the first (the joined table between the Flights and Weather Observations), applies the Random Forest model and returns the evaluation metrics.
The Random Forest model takes as features the flight information (as described on chapter 4) and the weather observations from n-hours before the scheduled departure and arrival time. The 'n-hours' both before scheduled departure and arrival can be selected by the user. Note that in the code, a time slot of "0" corresponds to the use of weather data from the same time slot as the forecast departure (or arrival).
The application also allows the user to input the delay threshold in minutes (15, 30, 60 etc.) that is used to build the target of our model (figure 38).
It's important to note that due to the limited computing power available with the cluster, we were able to run the model with weather observations only up to three hours before departure and arrival times.

Listing 2: Spark-submit command delay-prediction

```
spark-submit \
    --class com.DelayPredictionApplication \
    --master yarn \
    --num-executors 10 \
    --executor-cores 5 \
    --executor-memory 8G \
    --driver-memory 4G \
    --conf spark.dynamicAllocation.enabled=true \
    --conf spark.executor.memoryOverhead=2G \
    delaypredictionapp_2.12-0.1.jar
```



```
===================================
Welcome to the Delay Prediction App
===================================
Instructions:
* Enter the root path for your processed data.
   If you did not process data, you can use already processed data from:
  /students/iasd_20222023/asakhraoui/output

Current directory is:
/opt/cephfs/users/students/iasd_20222023/asakhraoui
Enter your processed data path:
/students/iasd_20222023/asakhraoui/output
Processed data path: /students/iasd_20222023/asakhraoui/output
/students/iasd_20222023/asakhraoui/output
Enter the time range before CRS (between 0 and 11 included):
5
Enter the time range after Scheduled Arrival (between 0 and 11 included):
5
Enter the delay threshold in minutes (between 15 and 60):
15
Enter the number of trees for RandomForest (recommended: 50):
50
Origin time range considered: 5
Destination time range considered: 5
Delay threshold: 15
Number of Trees for RandomForest: 50
```

Figure 38: Input required by the program - delay-prediction

5. Conclusion

In this project we had the opportunity to implement the approach outlined in "Using Scalable Data Mining for Predicting Flight Delays" by Belcastro et al. (2016) for flight delay prediction. The project encompassed the entire cycle, from data preparation to model evaluation.

In the preparation phase, we pre-process the Weather and Flight tables. This step was particularly complex for the Weather Observation table due to:

- the handling of missing values (we experimented two approaches, one to remove them, the other one to replace them with the daily average);
- the handling of multiple values for the same time;
- the handling of features such as 'sky condition' and 'weather type', since they are categorical with many values, which make the model more complex;
- the choice of the attributes to select based on their impact on flight delay and, therefore, are more relevant to the final model.

The join phase proved to be critical, demanding focused attention on partitioning and the evaluation of different approaches to the join operation. We have experimented two different methods: one based on a conditional join, the other reflecting the methodology presented in the article. The latter, was the one registering better performance in terms of computation time.

The results of our study revealed that adding weather information improve the prediction, but with not such a significant difference with the data without weather observations. We assume that some possible explanations of why the model performance does not improve as expected could be:

- the data at our disposal are not enough to provide an accurate model;
- the computing power of the resources available for this project were limited, implying the impossibility to fine tune the hyperparameters;
- the choice of the relevant weather attributes to feed into the model which may not be optimal.

We believe the above points represent directions for a future potential development of our project.

Appendix

.1. Code for conditional join method

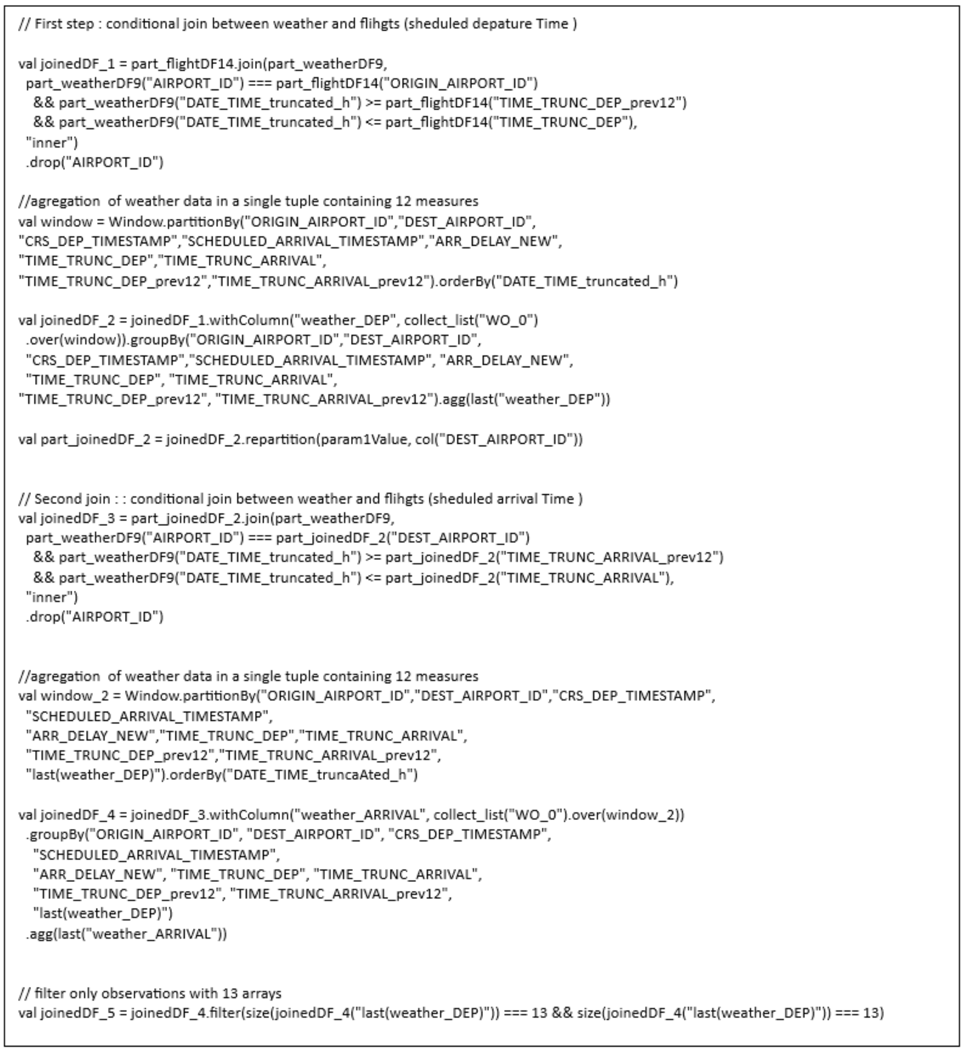
In Figure 39 shows the code that has been used to create the conditional join.

.2. Alternative code for replacing missing numerical data in the weather table with daily averages

Figure 40 shows the code that has been used to deal with missing numerical values on the weather table. Missing values have been replaced by the daily average.

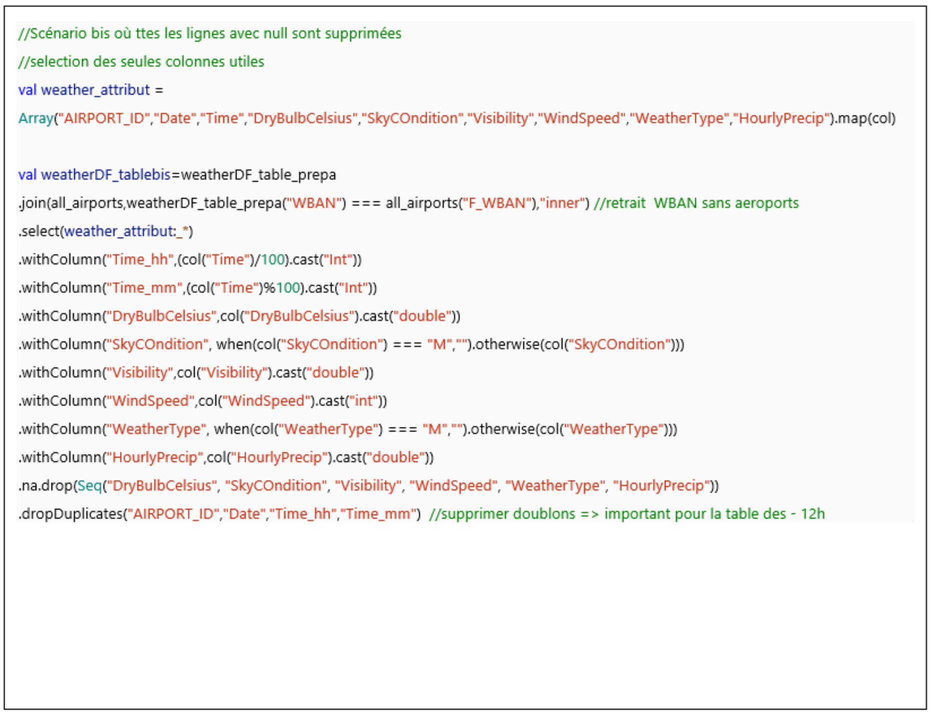
.3. Model results for different dataset sizes and weather hours

Figure 41 and 42 shows the results for tests carried out with 50 and 25 percent of the dataset, using a random split between train and test.



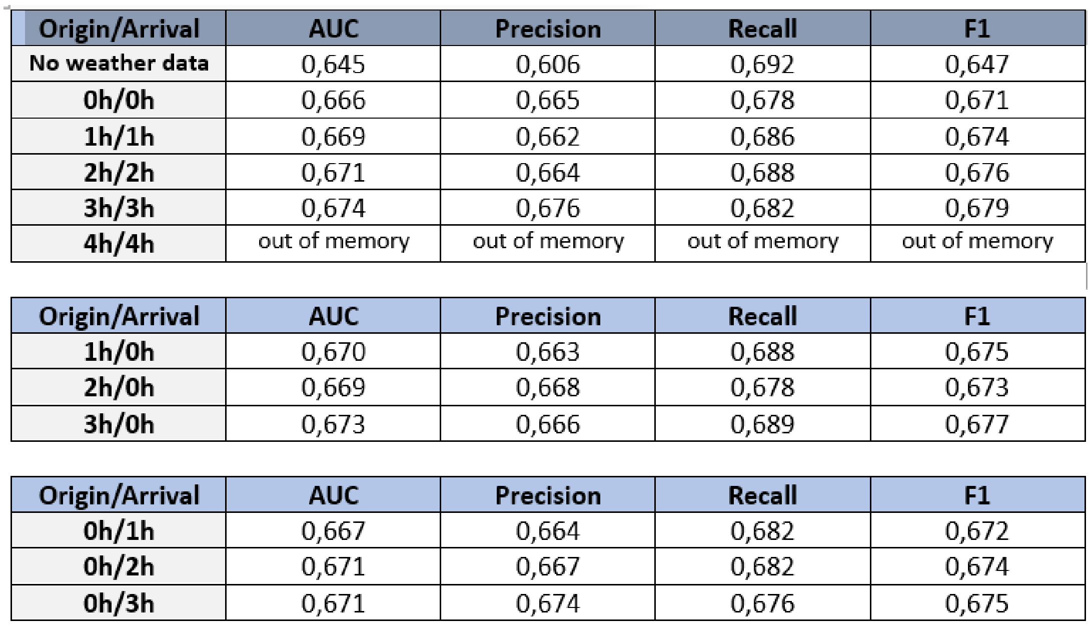
```
// First step : conditional join between weather and flihgts (sheduled depature Time )

val joinedDF_1 = part_flightDF14.join(part_weatherDF9,
  part_weatherDF9("AIRPORT_ID") === part_flightDF14("ORIGIN_AIRPORT_ID")
    && part_weatherDF9("DATE_TIME_truncated_h") >= part_flightDF14("TIME_TRUNC_DEP_prev12")
    && part_weatherDF9("DATE_TIME_truncated_h") <= part_flightDF14("TIME_TRUNC_DEP"),
  "inner")
  .drop("AIRPORT_ID")

//agregation  of weather data in a single tuple containing 12 measures
val window = Window.partitionBy("ORIGIN_AIRPORT_ID","DEST_AIRPORT_ID",
"CRS_DEP_TIMESTAMP","SCHEDULED_ARRIVAL_TIMESTAMP","ARR_DELAY_NEW",
"TIME_TRUNC_DEP","TIME_TRUNC_ARRIVAL",
"TIME_TRUNC_DEP_prev12","TIME_TRUNC_ARRIVAL_prev12").orderBy("DATE_TIME_truncated_h")

val joinedDF_2 = joinedDF_1.withColumn("weather_DEP", collect_list("WO_0")
  .over(window)).groupBy("ORIGIN_AIRPORT_ID","DEST_AIRPORT_ID",
  "CRS_DEP_TIMESTAMP","SCHEDULED_ARRIVAL_TIMESTAMP", "ARR_DELAY_NEW",
  "TIME_TRUNC_DEP", "TIME_TRUNC_ARRIVAL",
"TIME_TRUNC_DEP_prev12", "TIME_TRUNC_ARRIVAL_prev12").agg(last("weather_DEP"))

val part_joinedDF_2 = joinedDF_2.repartition(param1Value, col("DEST_AIRPORT_ID"))


// Second join : : conditional join between weather and flihgts (sheduled arrival Time )
val joinedDF_3 = part_joinedDF_2.join(part_weatherDF9,
  part_weatherDF9("AIRPORT_ID") === part_joinedDF_2("DEST_AIRPORT_ID")
    && part_weatherDF9("DATE_TIME_truncated_h") >= part_joinedDF_2("TIME_TRUNC_ARRIVAL_prev12")
    && part_weatherDF9("DATE_TIME_truncated_h") <= part_joinedDF_2("TIME_TRUNC_ARRIVAL"),
  "inner")
  .drop("AIRPORT_ID")


//agregation  of weather data in a single tuple containing 12 measures
val window_2 = Window.partitionBy("ORIGIN_AIRPORT_ID","DEST_AIRPORT_ID","CRS_DEP_TIMESTAMP",
  "SCHEDULED_ARRIVAL_TIMESTAMP",
  "ARR_DELAY_NEW","TIME_TRUNC_DEP","TIME_TRUNC_ARRIVAL",
  "TIME_TRUNC_DEP_prev12","TIME_TRUNC_ARRIVAL_prev12",
  "last(weather_DEP)").orderBy("DATE_TIME_truncaAted_h")

val joinedDF_4 = joinedDF_3.withColumn("weather_ARRIVAL", collect_list("WO_0").over(window_2))
  .groupBy("ORIGIN_AIRPORT_ID", "DEST_AIRPORT_ID", "CRS_DEP_TIMESTAMP",
    "SCHEDULED_ARRIVAL_TIMESTAMP",
    "ARR_DELAY_NEW", "TIME_TRUNC_DEP", "TIME_TRUNC_ARRIVAL",
    "TIME_TRUNC_DEP_prev12", "TIME_TRUNC_ARRIVAL_prev12",
    "last(weather_DEP)")
  .agg(last("weather_ARRIVAL"))


// filter only observations with 13 arrays
val joinedDF_5 = joinedDF_4.filter(size(joinedDF_4("last(weather_DEP)")) === 13 && size(joinedDF_4("last(weather_DEP)")) === 13)
```

Figure 39: Code for conditional join method



```
//Scénario bis où ttes les lignes avec null sont supprimées
//selection des seules colonnes utiles
val weather_attribut =
Array("AIRPORT_ID","Date","Time","DryBulbCelsius","SkyCOndition","Visibility","WindSpeed","WeatherType","HourlyPrecip").map(col)

val weatherDF_tablebis=weatherDF_table_prepa
.join(all_airports,weatherDF_table_prepa("WBAN") === all_airports("F_WBAN"),"inner") //retrait  WBAN sans aeroports
.select(weather_attribut:_*)
.withColumn("Time_hh",(col("Time")/100).cast("Int"))
.withColumn("Time_mm",(col("Time")%100).cast("Int"))
.withColumn("DryBulbCelsius",col("DryBulbCelsius").cast("double"))
.withColumn("SkyCOndition", when(col("SkyCOndition") === "M","").otherwise(col("SkyCOndition")))
.withColumn("Visibility",col("Visibility").cast("double"))
.withColumn("WindSpeed",col("WindSpeed").cast("int"))
.withColumn("WeatherType", when(col("WeatherType") === "M","").otherwise(col("WeatherType")))
.withColumn("HourlyPrecip",col("HourlyPrecip").cast("double"))
.na.drop(Seq("DryBulbCelsius", "SkyCOndition", "Visibility", "WindSpeed", "WeatherType", "HourlyPrecip"))
.dropDuplicates("AIRPORT_ID","Date","Time_hh","Time_mm")  //supprimer doublons => important pour la table des - 12h
```

Figure 40: Alternative code for replacing missing numerical data in the weather

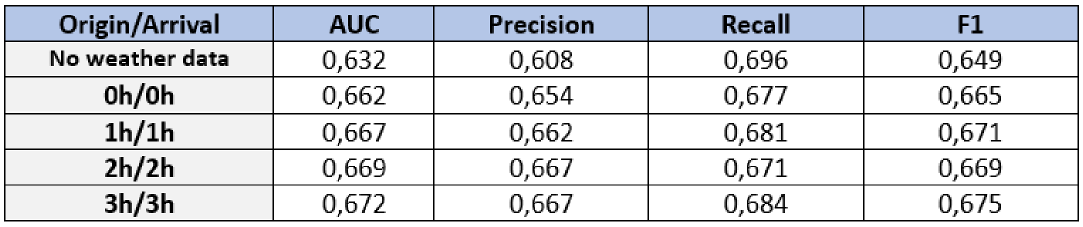


Origin/Arrival	AUC	Precision	Recall	F1
No weather data	0,645	0,606	0,692	0,647
0h/0h	0,666	0,665	0,678	0,671
1h/1h	0,669	0,662	0,686	0,674
2h/2h	0,671	0,664	0,688	0,676
3h/3h	0,674	0,676	0,682	0,679
4h/4h	out of memory	out of memory	out of memory	out of memory

Origin/Arrival	AUC	Precision	Recall	F1
1h/0h	0,670	0,663	0,688	0,675
2h/0h	0,669	0,668	0,678	0,673
3h/0h	0,673	0,666	0,689	0,677

Origin/Arrival	AUC	Precision	Recall	F1
0h/1h	0,667	0,664	0,682	0,672
0h/2h	0,671	0,667	0,682	0,674
0h/3h	0,671	0,674	0,676	0,675

Figure 41: Results using 50 percent of the dataset



Origin/Arrival	AUC	Precision	Recall	F1
No weather data	0,632	0,608	0,696	0,649
0h/0h	0,662	0,654	0,677	0,665
1h/1h	0,667	0,662	0,681	0,671
2h/2h	0,669	0,667	0,671	0,669
3h/3h	0,672	0,667	0,684	0,675

Figure 42: Results using 25 percent of the dataset