2017 Edition

In this notebook, we will learn how to solve the regression problem of predicting flight delays, using decision trees and random forests.

Goals

The main goals of this project are the following:

1. Revisit the concepts behind Decision Trees and Random Forests
2. Build a simple methodology to address Data Science projects
3. Use the existing implementation of Random Forests in MLLib in a specific use case, that is to predict the delay of flights

Steps

- First, in section 1, we will go through a short introduction about the fundamentals of Decision Trees and Random Forests, such as feature definition, the form of a decision tree, how does it work and the idea of a forest of decision trees. If the student is familiar with these topics, skip to section 2.
- In section 2, we delve into the details of the use case of this notebook including: providing the context, introducing the data and the basic methodology to address the project in this notebook
- In section 3, we perform data exploration
- In section 4, we build the statistical model and validate it

1. Decision trees and Random Forests: Simple but Powerful Algorithms

> Prediction is very difficult, especially if it’s about the future. (Niels Bohr)

Decision trees are a very popular approach to prediction problems. Decision trees can be trained from both categorical and numerical features, to perform classification and regression. They are the oldest and most well-studied types of predictive analytics. In many analytics packages and libraries, most algorithms are devoted either to address classification or regression problems, and they include for example support vector machines (SVM), neural networks, naïve Bayes, logistic regression, and deep learning...

In general, classification refers to the problem of predicting a label, or category, like *spam/not spam*, *rainy/sunny/mild*, for some given data. Regression refers to predicting a numeric quantity like salary, temperature, delay time, product's price. Both classification and regression involve predicting one (or more) values given one (or more) other input values. They require labelled data to perform a training phase, which builds the statistical model: they belong to *supervised learning* techniques.

1.1 Feature definition

To understand how regression and classification operate, it is necessary to briefly define the terms that describe their input and output.

Assume that we want to predict the temperature of tomorrow given today's weather information. The weather information is a loose concept. For example, we can use many variables to express today's weather such as:

- the average humidity today
- today's high temperature
- today's low temperature
- wind speed
- outlook: e.g. cloudy, rainy, or clear
- ....

These variables are called *features* or *dimensions*.

Each variable can be quantified. For example, high and low temperatures are measured in degrees Celsius, humidity can be measured as a fraction between 0 and 1, and weather type can be labeled `cloudy`, `rainy` or `clear`... So, the weather today can be expressed by a list of values: `11.4, 18.0, 0.64, 20, cloudy`. Each feature is also called a `predictor`. Together, they constitute a `feature vector`.

A feature whose domain is a set of categories is called **categorical feature**. In our example, `outlook` is a categorical feature. A feature whose values are numerical is called **numerical feature**. In our example, `temperature` is a numerical feature.

Finally, tomorrow's temperature, that is what we want to predict, is called *target feature*.

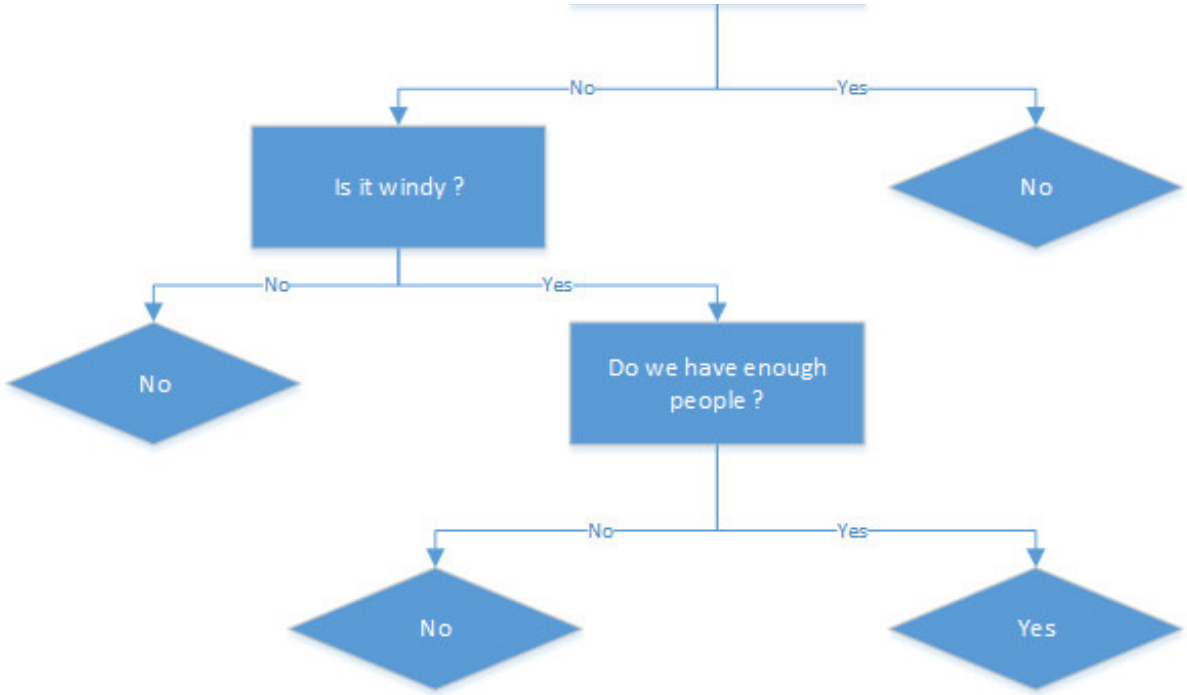
1.2 Decision Trees & Random Forests

The first question that you might ask is: "Why Decision trees and not another approach?"

Well, the literature shows that the family of algorithms known as decision trees can naturally handle both categorical and numeric features. The training process is easy to understand. The model is easy to interpret. They are robust to outliers in the data, meaning that a few extreme and possibly erroneous data points should not affect the tree at all. The model can be trained in parallel easily. The accuracy is comparable to other methods... In short, there are lots of advantages when using decision trees with respect to other methods!

The way we use a tree model is very simple to understand. We can say that this process "mimics" the way humans take decisions. For example, to decide whether to play football or not, a natural question would be "does it rain now?". If yes, the decision is `no`. If it's sunny, the condition is favorable to play football. A second natural question could be: "is it windy?". If no, then you may want to stay at home because otherwise it is going to be too hot. Otherwise, a third plausible question could be: "do we have enough people?". If no, then there's no point playing. Otherwise, time to play!

Using a decision tree allows to follow a similar process to that described above (see the image below). Given a new input, the algorithm traverses the tree in a such a way that the input satisfies the condition of each node until reaching a leaf one. The value of the leaf node is the decision.



The tree model in the figure is built from historical information concerning many past days. The `feature predictor` contains three features: Rain, Is_Windy, Enough_People. An example of the training data is as follows:

Rain	Is_Windy	Enough_People	Play
Yes	Yes	No	No
No	No	No	No
No	Yes	Yes	Yes
No	No	Yes	No

As you can see, in the training data, we know the values of predictors and we also know the corresponding answer: we have the ground truth.

One limitation of decision trees is that it's easy to incur in overfitting problems. In other words, the model is too fit to the training data, it is too precise and not general enough. So, when testing the quality of predictions with different testing sets, accuracy could fluctuate. To overcome this limitation, the tree can be pruned after it is built, or even be pruned during the training process. Another approach is building a `Random Decision Forest`.

A `Random Decision Forest`, as its name implies, is a forest of random Decision trees. Each tree element is built randomly from the training data. Randomization generally applies to:

- Building new training data: Random selection of samples from the training data (with replacement) from the original training data
- When building a node: Random selection of a subset of features

To take a decision, the forest "asks" all trees about their prediction, and then chooses the outcome which is the most voted.

2. Use case: Flights delay prediction

2.1 Context

Every day, in US, there are thousands of flights departures and arrivals: unfortunately, as you may have noticed yourself, flight delays are not a rare event!! Now, given historical data about flights in the country, including the delay information that was computed *a-posteriori* (so the ground truth is available), we want to build a model that can be used to predict how many minutes of delay a flight might experience in the future. This model should provide useful information for the airport to manage better its resources, to minimize the delays and their impact on the journey of its passengers. Alternatively, astute passengers could even use the model to choose the best time for flying, such as to avoid delays.

2.2 Data

The data we will use in this notebook has been collected by the RITA (Research and Innovative Technology Administration), and it contains details facets about each air flight that happened in the US between 1987 and 2008. It includes 29 variables such as the origin airport, the destination airport, the scheduled departure time, day, month, the arrival delay... For more information, please visit the following link, that provides a lot of detail on the data. Our goal is to build a model to predict `the arrival delay`.

2.3 Methodology

For our project, we can follow a simple methodology:

- Understand clearly the context, the data and the goal of the project
- Pre-process the data (data cleaning): the data can contain invalid values or missing values. We have to process our data to deal with them
- Retrieve descriptive information about data: the idea is to discover if whether the data has patterns, whether features have patterns, the skew of values...
- Select appropriate features: Only work with significant features will save us memory, communication cost, and ultimately, training time. Feature selection is also important as it can reduce the impact of noise that characterize the unimportant features.
- Divide the data into training and testing set
- Build a model from the feature in the training set
- Test the model

3. Let's play: Data Exploration

Now it's time to apply the simple methodology outlined in section 2.3 on the use case of this notebook.

Note: The source code in this lecture should be executed sequentially in the order.

3.1 Understanding the data schema

The data has 29 features, that can be either categorical or numerical. For example, the `src_airport` (source airport) is categorical: there exist no comparison operator between airport names. We can not say "SGN is bigger than NCE". The departure is numerical, for which a comparison operator exists.

For instance, "flight departing before 6PM" can be express by "departure_time < 1800".

In this use case, most features are numerical, except `carier`, `flight_number`, `cancelled`, `cancelation_code` and `diverted`.

The data contains a header, that is useless in building the statistical model. In addition, we already know the data schema, so we can safely neglect it. Note that there are some features with missing values in some lines of the dataset. The missing values are marked by "NA". These values can cause problems when processing and can lead to unexpected results. Therefore, we need to remove the header and replace all "NA" values by empty values, such as they can be interpreted as null values.

As we have seen already, there are multiple ways to manipulate data:

- Using the RDD abstraction
- Using the DataFrame abstraction. DataFrames can be thought of as distributed tables: each item is a list of values (the columns). Also, the value in each row of each column can be accessed by the column's name.

Next, we will focus on using DataFrames. However, to use DataFrames, the data must be clean (no invalid values). That means we cannot create DataFrame directly from the "RAW" data. Instead, we will first create an RDD from RAW data, produce a new, clean RDD, then transform it to a DataFrame and work on it. The RDD `cleaned_data` is an RDD[String]. We need to transform it to RDD[(TypeOfColumn1, TypeOfColumn2,.., TypeOfColumn29)] then call a function to create a DataFrame from the new RDD.

3.2 Data cleaning

Let's prepare for the cleaning step: Loading the data into an RDD.

First, we need to import some useful python modules for this notebook.

In [1]:

```
import os
import sys
import re
from pyspark import SparkContext
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.sql.functions import *
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import pyspark.sql.functions as func
import matplotlib.patches as mpatches

# to start testing, we can focus on a single year
input_path = "/datasets/airline/1994.csv"
raw_data = sc.textFile(input_path)
```

Question 1

Remove the header and replace the invalid values in our input dataset.

Question 1.1

> How many records (rows) in the RAW data?

In [2]:

```
print("number of rows before cleaning:", raw_data.count())
```

```
number of rows before cleaning: 5180049
```

Question 1.2

> Except for the first column, the others might contain missing values, which are denoted by `NA`. Remove the header and replace NA by an empty character. How many records are left after cleaning the RAW dataset? **NOTE**: be careful with the valid values that can contain string `NA` inside.

In [3]:

```
# extract the header
header = raw_data.first()

# replace invalid data with NULL and remove header
cleaned_data = (raw_data\
        # filter out the header
        .filter(lambda row:row!=header)
        # replace the missing values with empty characters
        .map(lambda row: row.replace(',NA', ','))
        )

print("number of rows after cleaning:", cleaned_data.count())
```

```
number of rows after cleaning: 5180048
```

3.3 Transforming our data to a DataFrame

Now the data is clean, valid and can be used to create DataFrame. First, we will declare the data schema for the DataFrame. By doing that, we can specify the name and data type of each column.

In [4]:

```
sqlContext = SQLContext(sc)


# Declare the data schema
# see http://stat-computing.org/dataexpo/2009/the-data.html
# for more information
airline_data_schema = StructType([ \
    #StructField( name, dataType, nullable)
    StructField("year",                    IntegerType(), True), \
```

```
    StructField("month",                    IntegerType(), True), \
    StructField("day_of_month",             IntegerType(), True), \
    StructField("day_of_week",              IntegerType(), True), \
    StructField("departure_time",           IntegerType(), True), \
    StructField("scheduled_departure_time", IntegerType(), True), \
    StructField("arrival_time",             IntegerType(), True), \
    StructField("scheduled_arrival_time",   IntegerType(), True), \
    StructField("carrier",                  StringType(),  True), \
    StructField("flight_number",            StringType(),  True), \
    StructField("tail_number",              StringType(), True), \
    StructField("actual_elapsed_time",      IntegerType(), True), \
    StructField("scheduled_elapsed_time",   IntegerType(), True), \
    StructField("air_time",                 IntegerType(), True), \
    StructField("arrival_delay",            IntegerType(), True), \
    StructField("departure_delay",          IntegerType(), True), \
    StructField("src_airport",              StringType(),  True), \
    StructField("dest_airport",             StringType(),  True), \
    StructField("distance",                 IntegerType(), True), \
    StructField("taxi_in_time",             IntegerType(), True), \
    StructField("taxi_out_time",            IntegerType(), True), \
    StructField("cancelled",                StringType(),  True), \
    StructField("cancellation_code",        StringType(),  True), \
    StructField("diverted",                 StringType(),  True), \
    StructField("carrier_delay",            IntegerType(), True), \
    StructField("weather_delay",            IntegerType(), True), \
    StructField("nas_delay",                IntegerType(), True), \
    StructField("security_delay",           IntegerType(), True), \
    StructField("late_aircraft_delay",      IntegerType(), True)\
])
```

To "convert" an RDD to DataFrame, each element in the RDD must be a list of column values that match the data schema.

In [5]:

```
# convert each line into a tuple of features (columns)
cleaned_data_to_columns = cleaned_data.map(lambda l: l.split(","))\
    .map(lambda cols:
         (
            int(cols[0])  if cols[0] else None,
            int(cols[1])  if cols[1] else None,
            int(cols[2])  if cols[2] else None,
            int(cols[3])  if cols[3] else None,
            int(cols[4])  if cols[4] else None,
            int(cols[5])  if cols[5] else None,
            int(cols[6])  if cols[6] else None,
            int(cols[7])  if cols[7] else None,
            cols[8]       if cols[8] else None,
            cols[9]       if cols[9] else None,
            cols[10]      if cols[10] else None,
            int(cols[11]) if cols[11] else None,
            int(cols[12]) if cols[12] else None,
            int(cols[13]) if cols[13] else None,
            int(cols[14]) if cols[14] else None,
            int(cols[15]) if cols[15] else None,
            cols[16]      if cols[16] else None,
            cols[17]      if cols[17] else None,
            int(cols[18]) if cols[18] else None,
            int(cols[19]) if cols[19] else None,
```

```
            int(cols[20]) if cols[20] else None,
            cols[21]      if cols[21] else None,
            cols[22]      if cols[22] else None,
            cols[23]      if cols[23] else None,
            int(cols[24]) if cols[24] else None,
            int(cols[25]) if cols[25] else None,
            int(cols[26]) if cols[26] else None,
            int(cols[27]) if cols[27] else None,
            int(cols[28]) if cols[28] else None
        ))
```

To train our model, we use the following features: `year`, `month`, `day_of_month`, `day_of_week`, `scheduled_departure_time`, `scheduled_arrival_time`, `arrival_delay`, `distance`, `src_airport`, `dest_airport`.

Question 2

From RDD `cleaned_data_to_columns` and the schema `airline_data_schema` which are declared before, create a new DataFrame **`df`**. Note that, we should only select the necessary features defined above: [`year`, `month`, `day_of_month`, `day_of_week`, `scheduled_departure_time`, `scheduled_arrival_time`, `arrival_delay`, `distance`, `src_airport`, `dest_airport`]. Finally, the data should be cached.

In [6]:

```
# create dataframe df
df =
(sqlContext.createDataFrame(cleaned_data_to_columns,airline_data_schema)
        .select(['year', 'month', 'day_of_month', 'day_of_week', 'scheduled_
departure_time'\
                 , 'scheduled_arrival_time','carrier', 'arrival_delay', 'dis
tance', 'src_airport', 'dest_airport'])
        .cache()
    )
```

3.4 Descriptive statistics

Next, we will go over a series of simple queries on our data, to explore it and compute statistics. These queries directly map to the questions you need to answer.

NOTE: finding the right question to ask is difficult! Don't be afraid to complement the questions below, with your own questions that, in your opinion, are valuable ways to inspect data. This can give you extra points!

- Basic queries:
  - How many unique origin airports?
  - How many unique destination airports?
  - How many carriers?
  - How many flights that have a scheduled departure time later than 18h00?
- Statistic on flight volume: this kind of statistics are helpful to reason about delays. Indeed, it is plausible to assume that "*the more flights in an airport, the higher the probability of delay*"

is plausible to assume that *the more flights in an airport, the higher the probability of delay*?
    - How many flights in each month of the year?
    - Is there any relationship between the number of flights and the days of week?
    - How many flights in different days of months and in different hours of days?
    - Which are the top 20 busiest airports (this depends on inbound and outbound traffic)?
    - Which are the top 20 busiest carriers?
- Statistic on the fraction of delayed flights
    - What is the percentage of delayed flights (over total flights) for different hours of the day?
    - Which hours of the day are characterized by the longest flight delay?
    - What are the fluctuation of the percentage of delayed flights over different time granularities?
    - What is the percentage of delayed flights which depart from one of the top 20 busiest airports?
    - What is the percentage of delayed flights which belong to one of the top 20 busiest carriers?

Question 3: Basic queries

Question 3.1

How many origin airports? How many destination airports?

In [7]:

```
num_src_airport = df.select(['src_airport']).distinct().count()
num_dest_airport = df.select(['dest_airport']).distinct().count()
print("number of origin airports ", num_src_airport)
print("number of destination airports ", num_dest_airport)
```

```
number of origin airports  224
number of destination airports  225
```

Question 3.2

How many carriers?

In [8]:

```
num_carrier = df.select(['carrier']).distinct().count()
print("the number distinct carriers:", num_carrier)
```

```
the number distinct carriers: 10
```

Question 3.3

How many night flights (that is, flights departing later than 6pm)?

In [9]:

```
print("the number of night flights:", df[df.scheduled_departure_time > 1800
].count())
```

```
the number of night flights: 1078203
```

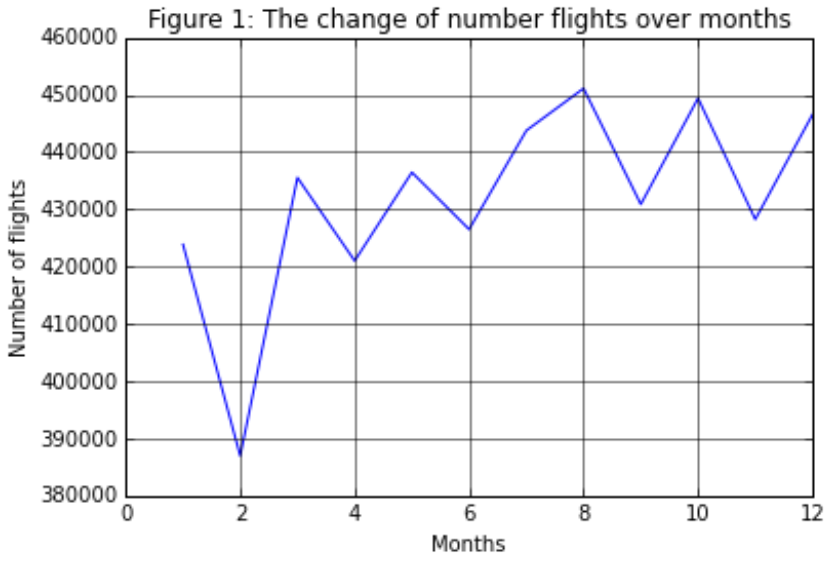
Question 4: Flight volume statistics

Question 4.1:

> How many flights in each month of the year? Plot the changes over months by a line chart and comment the figure. From the result, we can learn the dynamics of flight volume over months. For example, if we only consider flights in 1994 (to start, it's always better to focus on smaller amount of data), we can discuss about which months are most likely to have flights experiencing delays.

In [10]:

```
statistic_month = df.groupby('month').count().sort('month').collect()
#statistic_day_of_week.show()
pdf = pd.DataFrame(data=statistic_month)
plt.xlabel("Months")
plt.ylabel("Number of flights")
plt.title('Figure 1: The change of number flights over months')
plt.grid(True,which="both",ls="-")
plt.plot(pdf[0], pdf[1])
plt.show()
```



Comment

The number of flights is at its maximum during the months of August and October and at its lowest during the month of February.

These observations are aligned with the school holidays: so we have a high number of flights at

each school break (summer, christmas..)

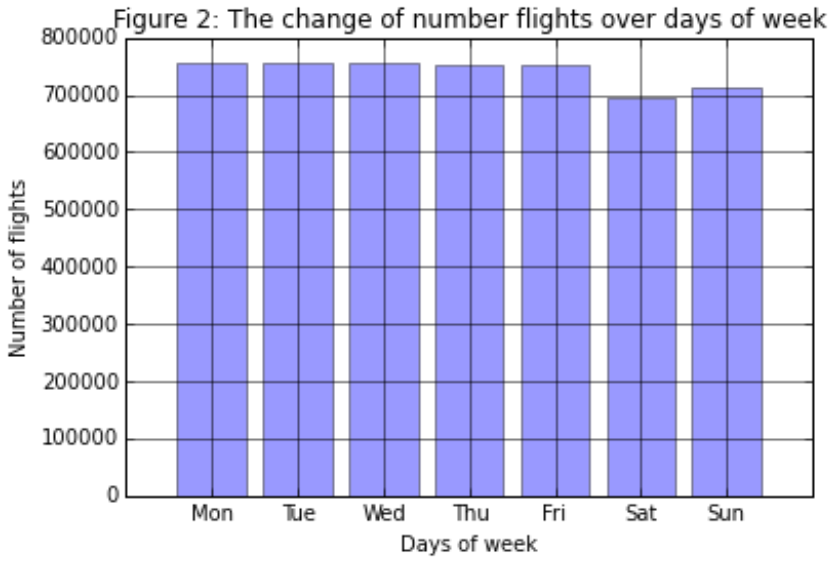
In february, there is a low demand for plane tickets (low season).

Question 4.2:

Is there any relationship between the number of flights and the days of the week? Plot a bar chart and interpret the figure. By answering this question, we could learn about the importance of the weekend/weekday feature for our predictive task.

In [11]:

```
statistic_day_of_week =
df.groupby('day_of_week').count().sort('day_of_week').collect()
pdf = pd.DataFrame(data=statistic_day_of_week)
plt.xlabel("Days of week")
plt.ylabel("Number of flights")
plt.title('Figure 2: The change of number flights over days of week')
plt.grid(True,which="both",ls="-")
map_int_into_day = { 1:"Mon", 2:"Tue", 3:"Wed", 4:"Thu", 5:"Fri", 6:"Sat", 7
:"Sun" }
day_of_week_label = pdf[0].map(lambda i: map_int_into_day[i])

# plot bar chart
plt.bar(pdf[0],pdf[1], align='center', alpha=0.4)
plt.xticks(pdf[0], day_of_week_label)
plt.show()
```



Comment

The number of flights per day of a week is (almost) steady. It's a bit lower in saturday, probably because people travelling during the weekend travel on friday and come back on sunday or monday morning.

In addition, people are working less on weekends so there are no business passengers so less need for flights

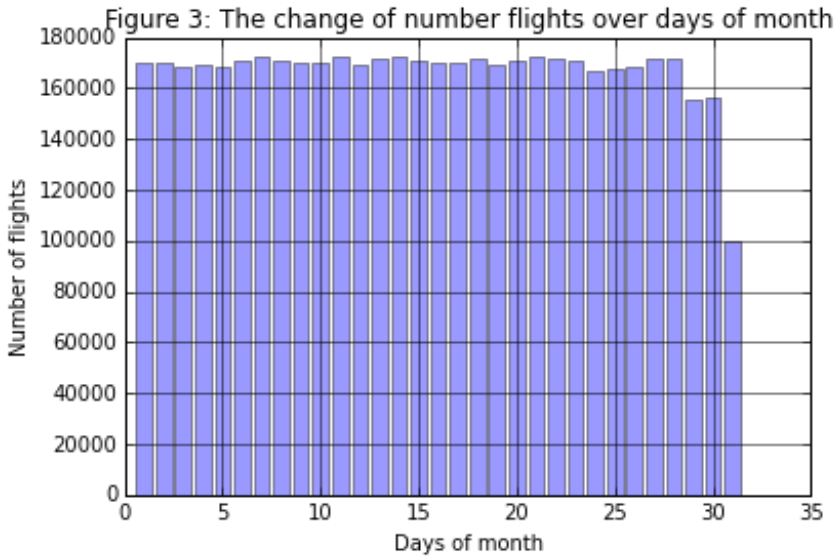
need for flights.

Question 4.3

How many flights in different days of months and in different hours of days? Plot bar charts, and interpret your figures.

In [12]:

```
df[df.scheduled_departure_time > 1800]

statistic_day_of_month = df.groupby('day_of_month').count().sort('day_of_month').collect()
pdf = pd.DataFrame(data=statistic_day_of_month)
plt.xlabel("Days of month")
plt.ylabel("Number of flights")
plt.title('Figure 3: The change of number flights over days of month')
plt.grid(True,which="both",ls="-")


# plot bar chart
plt.bar(pdf[0],pdf[1], align='center', alpha=0.4)
plt.show()
```



Comment

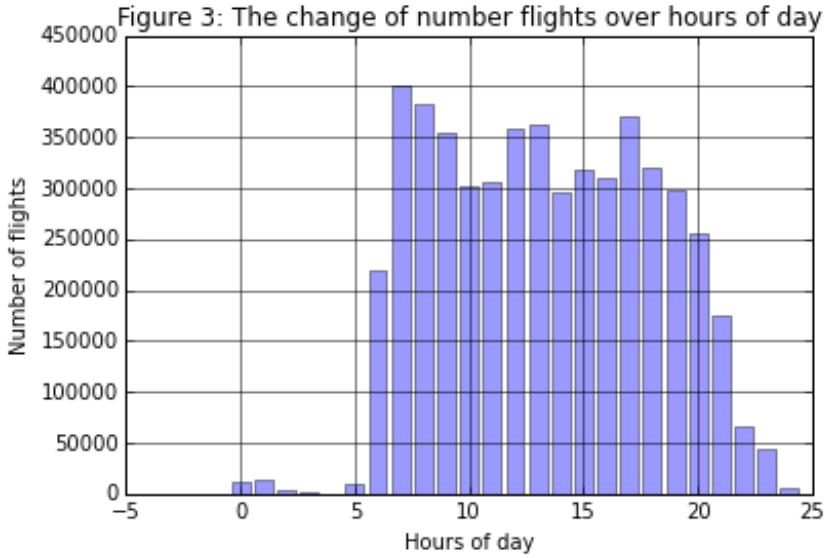
The number of flights over a month are steady. It doesn't fluctuate a lot except for 29,30 and 31.

For 31: it's because there are less months that has 31 days (only 7 out of 12).

For 29 and 30: it's because february was a normal year (not a leap year) so it had 28 days so there were only 11 months that had more than 30 days which proves that there should be a smaller number of flights in 29 and 30.

In [13]:

```
df_1= df.withColumn('hour', round(df.scheduled_departure_time/100, 0))
statistic_hour_of_day = df_1.groupby('hour').count().sort('hour').collect()
pdf = pd.DataFrame(data=statistic_hour_of_day)
plt.xlabel("Hours of day")
plt.ylabel("Number of flights")
plt.title('Figure 3: The change of number flights over hours of day')
plt.grid(True,which="both",ls="-")


# plot bar chart
plt.bar(pdf[0],pdf[1], align='center', alpha=0.4)
plt.show()
```



Comment

Most of the flights are between **7 AM** and **8 PM**.

There are not much flights **at night** (between 0AM and 5AM) probably because people prefer to travel in the morning so there is **less demand**, and also because of **noise restrictions** (many cities impose nighttime noise restrictions on airports, which would be violated by takeoffs). Moreover, Airplane equipment needs **maintenance** which is difficult to perform with heavy daytime plane utilization.

Flight numbers around **7 AM** and **8 AM** are higher because of **business passengers** and people who want to **arrive early**.

Question 4.4

Which are the **top 20** busiest airports: compute this in terms of aggregate inbound and outbound number of flights?

In [14]:

```
# consider outbound flights
stat_src = (df
            .groupBy(df.src_airport)
            .agg(func.count('*').alias('count1'))
            )

# consider inbound flights
stat_dest = (df
            .groupBy(df.dest_airport)
            .agg(func.count('*').alias('count2'))
            )


# full join the statistic of inbound flights and outbound flights
stat_airports = stat_src.join(stat_dest,stat_dest[0]==stat_src[0], how='ful
l')

# TOP 20 BUSIEST AIRPORTS
stat_airport_traffic = (stat_airports
                            # define the new column `total`
                            # which has values are equal to the sum of `cou
t1` and `count2`
                            .withColumn('total', stat_airports['count1'] + s
tat_airports['count2'])
                            # select top airpoint in termns of number of fl
ghts
                            .select(['src_airport','total']).orderBy(desc('
otal'))
                        )
stat_airport_traffic.show(20)
```

```
+-----------+------+
|src_airport| total|
+-----------+------+
|        ORD|561461|
|        DFW|516523|
|        ATL|443074|
|        LAX|306453|
|        STL|304409|
|        DEN|285526|
|        PHX|280560|
|        DTW|276272|
|        PIT|262939|
|        CLT|259712|
|        MSP|247980|
|        SFO|235478|
|        EWR|233991|
|        IAH|208591|
|        LGA|203362|
|        BOS|199696|
|        LAS|189920|
|        PHL|186897|
|        DCA|176115|
|        MCO|153720|
+-----------+------+
only showing top 20 rows
```

Question 4.5

Which are the **top 20** busiest carriers: compute this in terms of number of flights?

In [15]:

```
stat_carrier = (df
                .groupBy(df.carrier)
                .agg(func.count('*').alias('count'))
                .orderBy(desc('count'))
                )

stat_carrier.show(20)
```

```
+-------+------+
|carrier| count|
+-------+------+
|     DL|874526|
|     US|857906|
|     AA|722277|
|     UA|638750|
|     WN|565426|
|     CO|484834|
|     NW|482798|
|     TW|258205|
|     HP|177851|
|     AS|117475|
+-------+------+
```

Question 5

Statistics on the percentage of delayed flights

Question 5.1

What is the percentage of delayed flights for different hours of the day? Plot a bar chart and interpret the figure. **Remember** a flight is considered as delayed if it's actual arrival time is more than 15 minutes late than the scheduled arrival time.

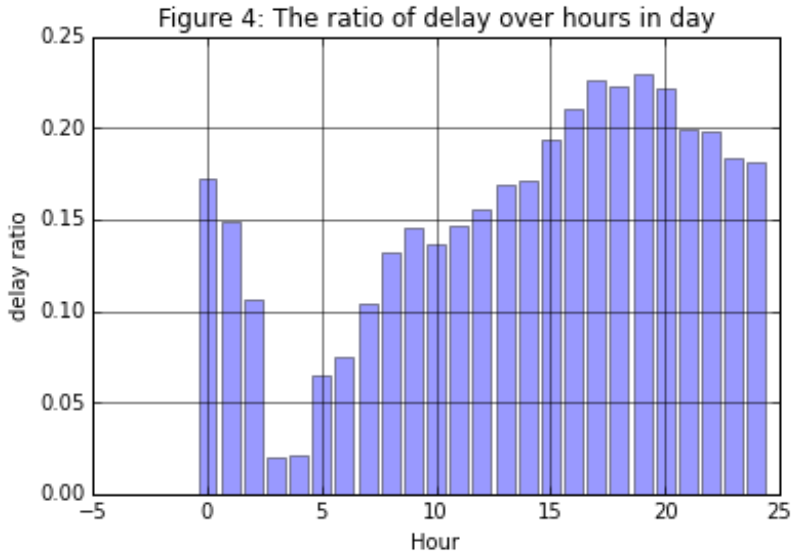
In [16]:

```
# create new column that marks whether the flights are delay
df_with_delay = df.withColumn('is_delay', when(df['arrival_delay'] >= 15, 1
).otherwise(0))

# create a new column that indicates the scheduled departure time in hour
# (ignore the part of minute)
delay_per_hour = df_with_delay.withColumn('hour',
round(df.scheduled_departure_time/100, 0))

# group by year and hour
statistic_delay_hour = delay_per_hour.groupBy(['year','hour'])

# calculate the delay ratio and create a new column
```

```
# calculate the delay ratio and create a new column
delay_ratio_per_hour = statistic_delay_hour.agg(
    (func.sum('is_delay')/func.count('*')).alias('delay_ratio')
)


# order the result by hour
delay_ratio_per_hour = (
            delay_ratio_per_hour
                .orderBy('hour')
                .select(['hour','delay_ratio']).collect()
    )

pdf_delay_ratio_per_hour = pd.DataFrame(data=delay_ratio_per_hour)
```

In [17]:

```
# plot a bar chart
plt.xlabel("Hour")
plt.ylabel("delay ratio")
plt.grid(True,which="both",ls="-")
plt.bar(pdf_delay_ratio_per_hour[0],pdf_delay_ratio_per_hour[1], align='center', alpha=0.4)
plt.title('Figure 4: The ratio of delay over hours in day')
plt.show()
```



Comment

The ratio of delay is higher between **5PM** and **8PM**.That's mainly because airports are more congested during certain times of day, and this may affect the delay distribution during a day.

The ratio of delay is lower at **3 AM** and **4AM**probably because there is less passengers in airports so flights are less inclined to be delayed and plane can even arrive in advance (earlier departure).

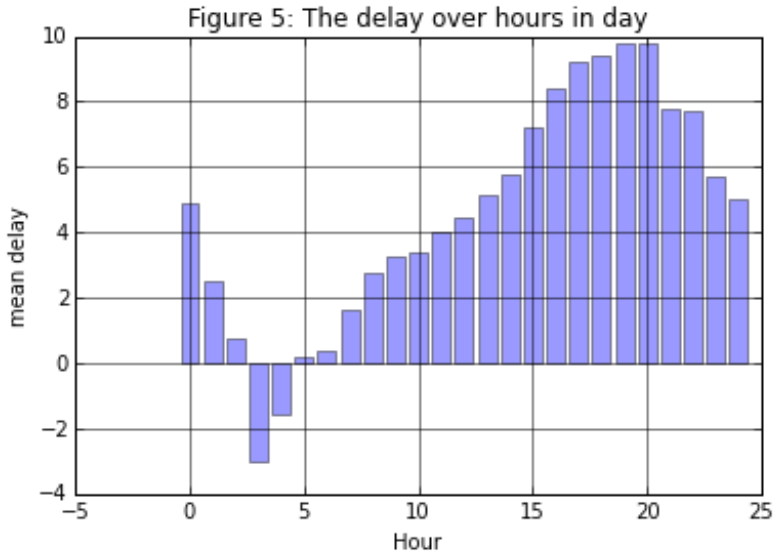
Question 5.2

You will realize that saying *"at 4 A.M. there is a very low chance of a flight being delayed"* is

not giving you a full picture of the situation. Indeed, it might be true that there is very little probability for an early flight to be delayed, but if it does, the delay might be huge, like 6 hours! Then, the question is: **which hours of the day are characterized by the largest delay?** Plot a Bar chart and explain it.

In [18]:

```
mean_delay_per_hour = statistic_delay_hour.agg(
    (func.mean('arrival_delay')).alias('mean_delay')
)

mean_delay_per_hour = (
    mean_delay_per_hour
        .orderBy('hour')
        .select(['hour','mean_delay']).collect()
    )

pdf_mean_delay_per_hour = pd.DataFrame(data=mean_delay_per_hour)

plt.xlabel("Hour")
plt.ylabel("mean delay")
plt.grid(True,which="both",ls="-")
plt.bar(pdf_mean_delay_per_hour[0],pdf_mean_delay_per_hour[1], align='center', alpha=0.4)
plt.title('Figure 5: The delay over hours in day')
plt.show()
```



Comment

From the curve we can see that some flights (between 2AM and 3AM) arrive earlier.

When comparing this plot, to the previous one, we can conclude that the hours that have a large ratio of delays also have the greater delay (and the other way around).

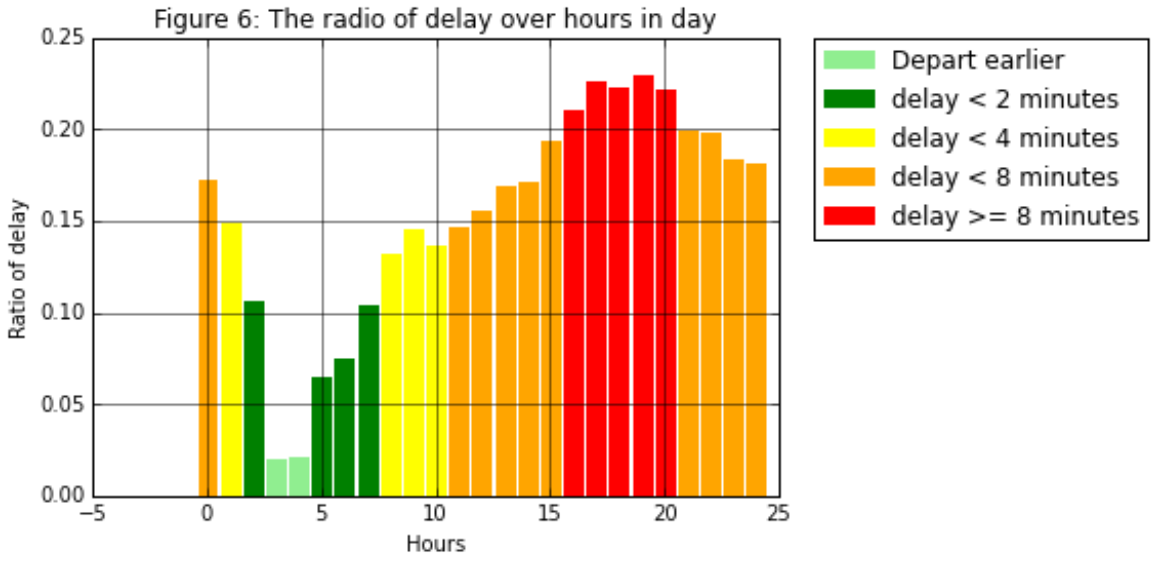
With data of year 1994, the flight from 3AM to 4AM often depart earlier than in their schedule. The

flights in the morning have less delay then in the afternoon and evening.

So, an attentive student should notice here that we have somehow a problem with the definition of delay! Next, we will improve how to represent and visualize data to overcome this problem.

In [19]:

```
#pdf2 = pd.DataFrame(data=mean_delay_per_hour.collect())
plt.xlabel("Hours")
plt.ylabel("Ratio of delay")
plt.title('Figure 6: The radio of delay over hours in day')
plt.grid(True,which="both",ls="-")
bars = plt.bar(pdf_delay_ratio_per_hour[0], pdf_delay_ratio_per_hour[1], al
ign='center', edgecolor = "black")
for i in range(0, len(bars)):
    color = 'red'
    if pdf_mean_delay_per_hour[1][i] < 0:
        color = 'lightgreen'
    elif pdf_mean_delay_per_hour[1][i] < 2:
        color = 'green'
    elif pdf_mean_delay_per_hour[1][i] < 4:
        color = 'yellow'
    elif pdf_mean_delay_per_hour[1][i] < 8:
        color = 'orange'

    bars[i].set_color(color)

patch1 = mpatches.Patch(color='lightgreen', label='Depart earlier')
patch2 = mpatches.Patch(color='green', label='delay < 2 minutes')
patch3 = mpatches.Patch(color='yellow', label='delay < 4 minutes')
patch4 = mpatches.Patch(color='orange', label='delay < 8 minutes')
patch5 = mpatches.Patch(color='red', label='delay >= 8 minutes')

plt.legend(handles=[patch1, patch2, patch3, patch4, patch5], bbox_to_anchor=
(1.05, 1), loc=2, borderaxespad=0.)

plt.show()
```



In the new figure (Figure 6), we have more information in a single plot. The flights in 3AM to 4AM have very low probability of being delayed, and actually depart earlier than their schedule. In contrast, the flights in the 4PM to 8PM range have higher chances of being delayed: in more than 50% of the cases, the delay is 8 minutes or more

cases, the delay is 5 minutes or more.

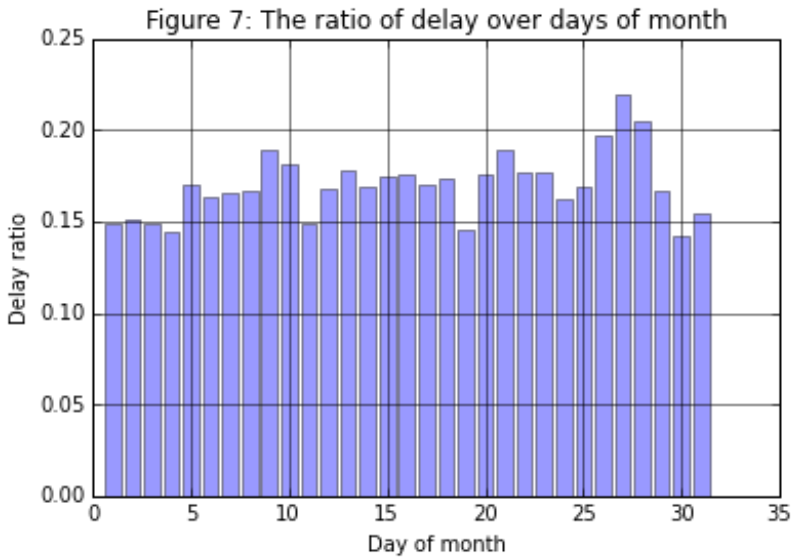
This example shows us that the way representing results are also important.

Question#### Question 5.3

> Plot a bar chart to show the percentage of delayed flights over days in a month

In [20]:

```
####### The changes of delay ratio over days of month ########
# calculate the delay ratio in each day of month
statistic_day_of_month = (
    df_with_delay
        .groupBy(df.day_of_month)
        .agg(func.sum('is_delay')/func.count('*')).alias('delay_ratio')
        # order by day_of_month
        .orderBy('day_of_month').collect()
)

# collect data and plot
pdf_day_of_month = pd.DataFrame(data=statistic_day_of_month)
plt.xlabel("Day of month")
plt.ylabel("Delay ratio")
plt.grid(True,which="both",ls="-")
plt.bar(pdf_day_of_month[0],pdf_day_of_month[1], align='center', alpha=0.4)
plt.title('Figure 7: The ratio of delay over days of month')
plt.show()
```



Comment

From the bar plot, we see that the ratio of delay over days of month is steady.

We also observe that it's a bit lower between first day of the month and the fourth, and also at the day 11,19,24 and at the end of the month (30 and 31).In all these days, there was one or more holiday, and more weekends days. So we can probably conclude that if there is less traffic there is probably less delay.

We see also that the delay is also higher between the 26th and the 28th. Probably because of

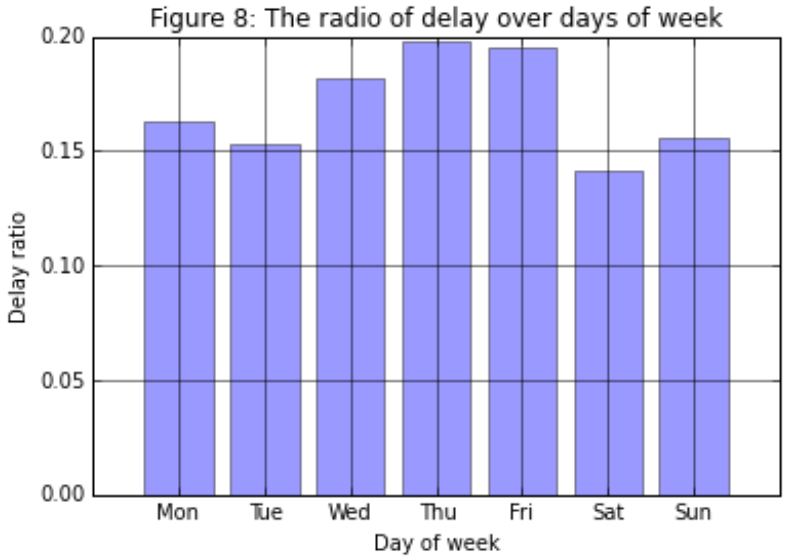
people travelling in christmas.

Question 5.4

Plot a bar chart to show the percentage of delayed flights over days in a week

In [21]:

```
####### The changes of delay ratio over days of week ########
# calculate the delay ratio in each day of week
statistic_day_of_week = (
    df_with_delay
        .groupBy(df.day_of_week)
        .agg(func.sum('is_delay')/func.count('*')).alias('delay_ratio')
        # order by day_of_week
        .orderBy('day_of_week').collect()
    )

# collect data and plot
pdf_day_of_week = pd.DataFrame(data=statistic_day_of_week)
map_int_into_day = { 1:"Mon", 2:"Tue", 3:"Wed", 4:"Thu", 5:"Fri", 6:"Sat", 7
:"Sun" }
day_of_week_label = pdf_day_of_week[0].map(lambda i: map_int_into_day[i])

plt.xlabel("Day of week")
plt.ylabel("Delay ratio")
plt.grid(True,which="both",ls="-")
plt.bar(pdf_day_of_week[0],pdf_day_of_week[1], align='center', alpha=0.4)
plt.title('Figure 8: The radio of delay over days of week')
plt.xticks(pdf_day_of_week[0], day_of_week_label)
plt.show()
```



Comment

The percentage of delay over the days of the week is almost steady

The percentage of delay over the days of the week is almost steady.

It's still slightly smaller at saturday and sunday because of less traffic.

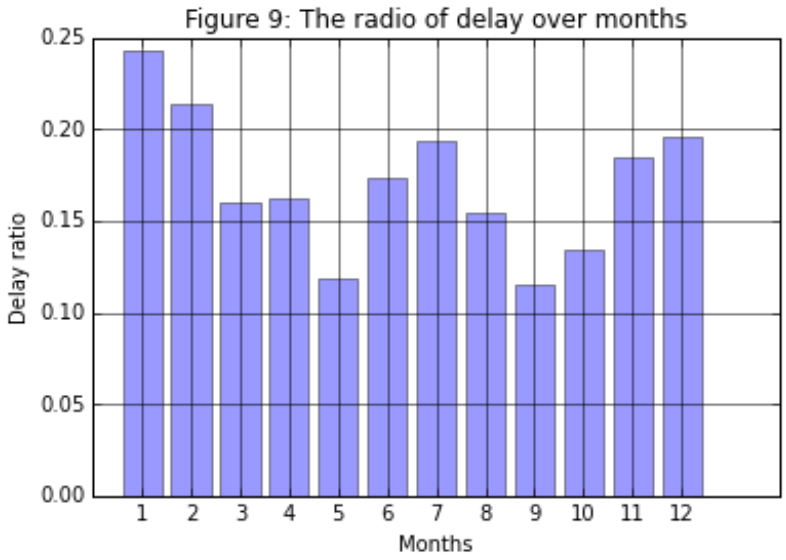
For some reason, there's more delay percentage at wednesday, thursday and friday than in monday and tuesday.

Question 5.5

Plot a bar chart to show the percentage of delayed flights over months in a year

In [22]:

```
####### The changes of delay ratio over months ########
# calculate the delay ratio in month
statistic_month = (
    df_with_delay
        .groupBy(df.month)
        .agg(func.sum('is_delay')/func.count('*')).alias('delay_ratio')
        # order by month
        .orderBy('month').collect()
    )

# collect data and plot
pdf_month = pd.DataFrame(data=statistic_month)
plt.xlabel("Months")
plt.ylabel("Delay ratio")
plt.grid(True,which="both",ls="-")
plt.bar(pdf_month[0],pdf_month[1], align='center', alpha=0.4)
plt.title('Figure 9: The radio of delay over months')
plt.xticks(pdf_month[0])
plt.show()
```



Comment

The percentage of delay is lower in May and September, and higher on December, January and February.

February.

September is the month when the summer break is over and people go back to work/studies so the airports are less crowded and flights are less inclined to make delays.

May is the last month of work/studies in the year so people are still busy with what they are doing. Hence the airports are less crowded.

A lot of people travel on late December and early january, so there are lot of delays on those period due to the crowds in airports, hence there are more delays.

We are ready now to draw some observations from our data, even if we have only looked at data coming from a year worth of flights:

- The probability for a flight to be delayed is low at the beginning or at the very end of a given months
- Flights on two first weekdays and on the weekend, are less likely to be delayed
- May and September are very good months for travelling, as the probability of delay is low (remember we're working on US data. Do you think this is also true in France?)

Putting things together, we can have a global picture of the whole year!

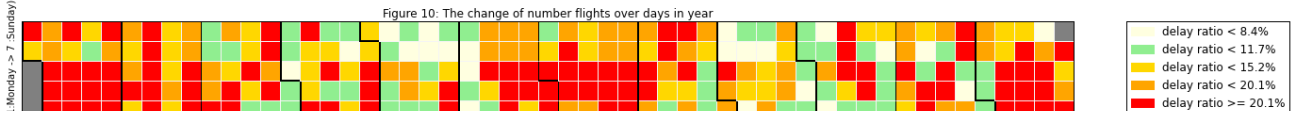
In [23]:

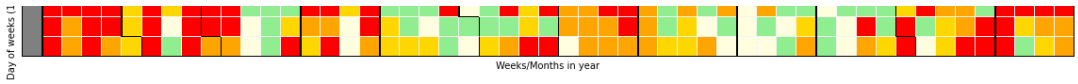
```
df_with_delay = df.withColumn('is_delay', when(df["arrival_delay"] >= 15, 1
).otherwise(0))
statistic_day = df_with_delay.groupBy(['year','month', 'day_of_month', 'day
_of_week'])\
    .agg((func.sum('is_delay')/func.count('*')).alias('delay_ratio'))

# assume that we do statistic on year 1994
statistic_day = statistic_day\
    .orderBy('year', 'month', 'day_of_month', 'day_of_week')
pdf = pd.DataFrame(data=statistic_day.collect())
```

In [24]:

```
fig = plt.figure(figsize=(20,10))

ax = fig.add_subplot(1,1,1)
plt.xlabel("Weeks/Months in year")
plt.ylabel("Day of weeks (1:Monday -> 7 :Sunday)")
plt.title('Figure 10: The change of number flights over days in year')

rec_size = 0.3
from matplotlib.patches import Rectangle
import datetime
num_days = len(pdf[0])
ax.patch.set_facecolor('gray')
ax.set_aspect('equal', 'box')
ax.xaxis.set_major_locator(plt.NullLocator())
ax.yaxis.set_major_locator(plt.NullLocator())

for i in range(0, num_days):
    # extract information from the result
    year = pdf[0][i]
    month = pdf[1][i]
    day of month = pdf[2][i]
```

```
    day_of_week = pdf[3][i]
    day_of_year= datetime.date(year=year, month=month, day=day_of_month).ti
metuple()
    week_of_year = datetime.date(year=year, month=month, day=day_of_month).
isocalendar()[1]

    # dealing with the week of the previous year
    if week_of_year == 52 and month == 1:
        week_of_year = 0

    # the coordinate of a day in graph
    X = week_of_year*rec_size
    Y = day_of_week*rec_size

    # use different colors to show the delay ratio
    color = 'white'
    if pdf[4][i] <= 0.084:
        color = 'lightyellow'
    elif pdf[4][i] <= 0.117:
        color = 'lightgreen'
    elif pdf[4][i] <= 0.152:
        color = 'gold'
    elif pdf[4][i] <= 0.201:
        color = 'orange'
    else:
        color = 'red'
    rect = plt.Rectangle((X - rec_size/2.0, Y - rec_size/2.0), rec_size, rec
_size,
                         alpha=1, facecolor=color, edgecolor='whitesmoke')

    ax.add_patch(rect)

    # drawing boders to separate months
    if day_of_month <= 7:
        rect2 = plt.Rectangle((X -rec_size/2.0, Y - rec_size/2.0), 0.01, rec
_size,
                         alpha=1, facecolor='black')
        ax.add_patch(rect2)
    if day_of_month == 1:
        rect2 = plt.Rectangle((X - rec_size/2.0, Y - rec_size/2.0), rec_size
, 0.01,
                         alpha=1, facecolor='black')
        ax.add_patch(rect2)
ax.autoscale_view()

patch1 = mpatches.Patch(color='lightyellow', label='delay ratio < 8.4%')
patch2 = mpatches.Patch(color='lightgreen', label='delay ratio < 11.7%')
patch3 = mpatches.Patch(color='gold', label='delay ratio < 15.2%')
patch4 = mpatches.Patch(color='orange', label='delay ratio < 20.1%')
patch5 = mpatches.Patch(color='red', label='delay ratio >= 20.1%')

plt.legend(handles=[patch1, patch2, patch3, patch4, patch5], bbox_to_anchor=
(1.05, 1), loc=2, borderaxespad=0.)

plt.show()
```





Question 5.6

Explain figure 10.

Comment

This data is a more general view of the delay ratio over the year. It contains the information in the questions above excluding the delay ratio per hours.

For instance, January and February (the two months on the left) are red which means that they is a high percentage of delay in them as we saw in 5.5.

We can see that there is a small percentage delay in tuesdays and that the rows from wednesday to friday contains a lot of red boxes which is consistent with the results of question 5.4.

Question 5.7

What is the delay probability for the top 20 busiest airports? By drawing the flight volume of each airport and the associated delay probability in a single plot, we can observe the relationship between airports, number of flights and the delay. HINT Function `.isin()` helps checking whether a value in column belongs to a list.

In [25]:

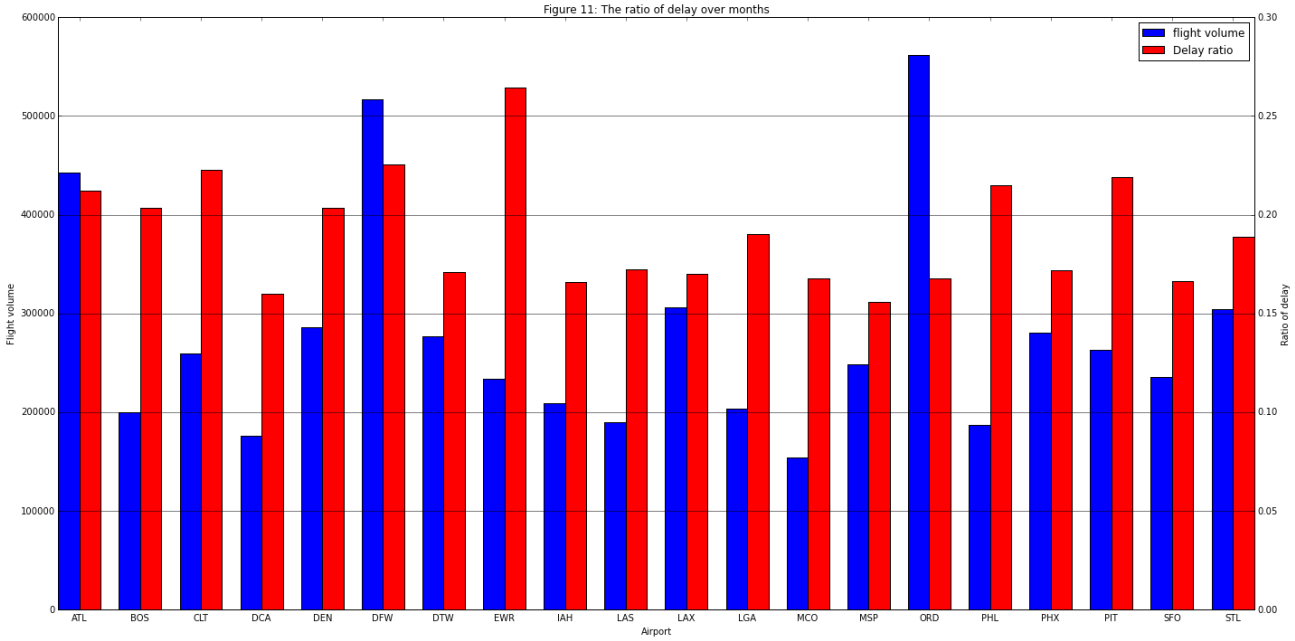
```
####### The delay ratio of the top 20 busiest airports ########
K = 20

# extract top_20_airports from stat_airport_traffic
top_20_airports = sorted([item[0] for item in stat_airport_traffic.take(K)]
)

# select the statistic of source airports
statistic_ratio_delay_airport = (
    df_with_delay
        # select only flights that depart from one of top 20 ariports
        .filter(df_with_delay.src_airport.isin(top_20_airports))
        # group by source airport
        .groupBy(['src_airport'])
        #  calculate the delay ratio
        .agg((func.sum('is_delay')/func.count('*')).alias('delay_ratio'))
        # sort by name of airport
        .orderBy(['src_airport'])
    )
statistic_ratio_delay_airport.show(20)
```

```
+-----------+-------------------+
|src_airport|        delay_ratio|
+-----------+-------------------+
|        ATL|0.21205403501801467|
|        BOS|0.20337767149902855|
|        CLT|0.22251161209048542|
|        DCA| 0.1599864322460286|
|        DEN|0.20354670607451195|
|        DFW|0.22524719636014578|
|        DTW|0.17069213736050923|
|        EWR|0.26439606741573035|
|        IAH| 0.1660171622737133|
|        LAS|0.17218759213241797|
|        LAX|0.16996104082244257|
|        LGA|0.19028312259483232|
|        MCO|  0.167725622406639|
|        MSP|0.15585690866890653|
|        ORD|0.16788302771286917|
|        PHL|0.21505583159694394|
|        PHX|0.17194317278139576|
|        PIT|0.21883994899867915|
|        SFO|0.16634949633351095|
|        STL|0.18877507271995725|
+-----------+-------------------+
```

In [26]:

```
# collect data and plot
pdf_ratio_delay_airport = pd.DataFrame(data=statistic_ratio_delay_airport.c
ollect())
pdf_top_20_airport_volume = pd.DataFrame(data=stat_airport_traffic.take(K),
columns=['src_airport', 'total'])
pdf_top_20_airport_volume = pdf_top_20_airport_volume.sort_values(by='src_a
irport')


#print(pdf_top_20_airport_volume)
index = np.arange(len(top_20_airports))
bar_width = 0.35
opacity = 0.4


fig = plt.figure(figsize=(20,10))

ax = fig.add_subplot(1,1,1)

ax2 = ax.twinx()
plt.axis('normal')
ax.set_xlabel("Airport")
ax.set_ylabel("Flight volume")
ax2.set_ylabel("Ratio of delay")
plt.xticks(index + bar_width, top_20_airports)
plt.title('Figure 11: The ratio of delay over months')
plt.grid(True,which="both",ls="-")
bar = ax.bar(index, pdf_top_20_airport_volume['total'],
        bar_width, color='b',
        label='flight volume')
bar2 = ax2.bar(index + 1.5*bar_width, pdf_ratio_delay_airport[1], bar_width
,
```

```
        align='center', color='r',
        label='Delay ratio')

lines, labels = ax.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc=0)

plt.tight_layout()
plt.show()
```



Comment

From the figure, the top 20 airports with the highest traffic volume (blue bar) and positively delayed flights (red bar) are shown. **ORD has the highest traffic volume, while the delay phenomenon is quite low (16.78%).**

Another airport needed to emphasize is EWR. **Although the traffic volume in EWR ranks 13 in the traffic volume, 26.5% of arriving flights from EWR are delayed.**

From this we can deduce that there is **no correlation** beetween the volume of flights over months and the corresponding delay, **we cannot say that a high volume implies a high delay**. But another parameter maybe the airport itself (if they have good and enough staff to handle the flights), we can confirm this by looking at the evolution of delay with number of staff team in the airport.

Question 5.8

What is the percentage of delayed flights which belong to one of the top 20 busiest carriers? Comment the figure!

In [27]:

```
K = 20
```

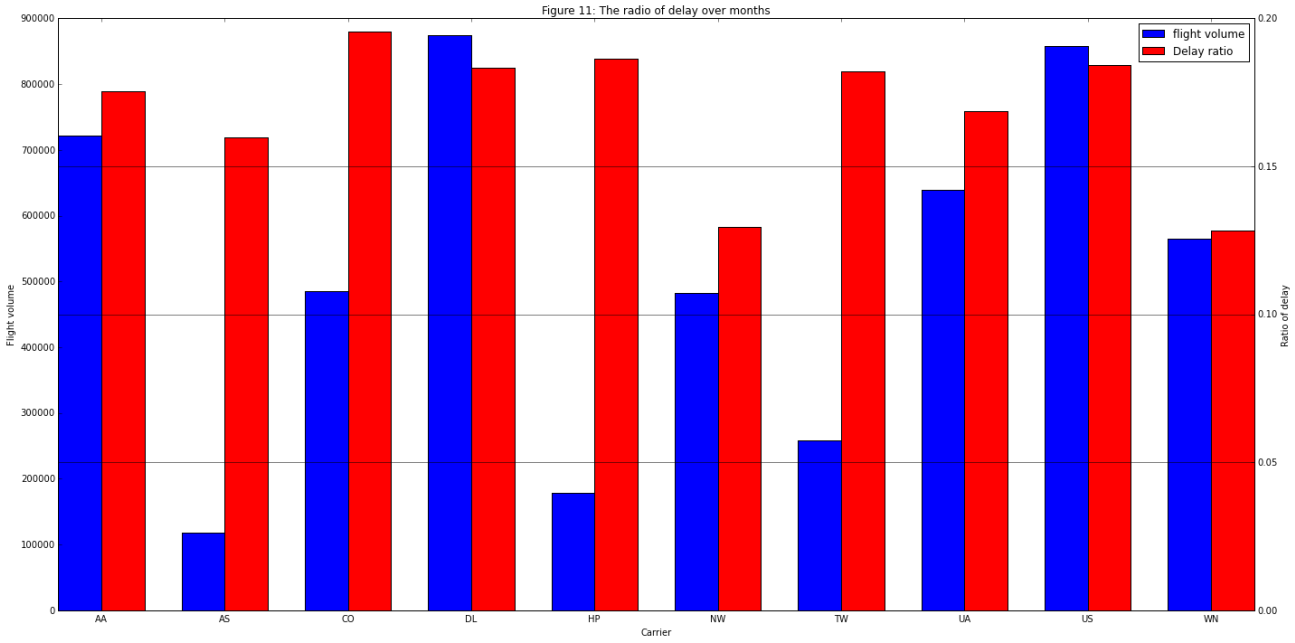
```
# extract top_20_carriers from stat_carrier
top_20_carriers = [item[0] for item in stat_carrier.take(K)]

statistic_ratio_delay_carrier = (
    df_with_delay
        # select only flights that belong from one of top 20 carriers
        .filter(df_with_delay.carrier.isin(top_20_carriers))
        # group by carriers
         .groupBy(['carrier'])
        #  calculate the delay ratio
        .agg((func.sum('is_delay')/func.count('*')).alias('delay_ratio'))
        # sort by name of airport
        .orderBy(['carrier'])
    )
statistic_ratio_delay_carrier.show(20)
```

```
+-------+-------------------+
|carrier|        delay_ratio|
+-------+-------------------+
|     AA| 0.1752444006939166|
|     AS| 0.1596424771227921|
|     CO| 0.1955576547849367|
|     DL|0.18328443065157582|
|     HP|0.18625141269939444|
|     NW| 0.1294806523639286|
|     TW|0.18212273193780135|
|     UA| 0.1686528375733855|
|     US|0.18422298014001534|
|     WN|0.12829795587751536|
+-------+-------------------+
```

In [28]:

```
# collect data and plot
pdf_ratio_delay_carrier = pd.DataFrame(data=statistic_ratio_delay_carrier.c
ollect())
pdf_top_20_carrier_volume = pd.DataFrame(data=stat_carrier.take(K), columns
=['carrier', 'count'])
pdf_top_20_carrier_volume = pdf_top_20_carrier_volume.sort_values(by='carri
er')
#print(pdf_top_20_carrier_volume)
top_20_carriers.sort()
index = np.arange(len(top_20_carriers))
bar_width = 0.35
opacity = 0.4


fig = plt.figure(figsize=(20,10))

ax = fig.add_subplot(1,1,1)

ax2 = ax.twinx()
plt.axis('normal')
ax.set_xlabel("Carrier")
ax.set_ylabel("Flight volume")
ax2.set_ylabel("Ratio of delay")
plt.xticks(index + bar_width, top_20_carriers)

plt.title('Figure 11. The ratio of delay over months')
```

```
plt.title('Figure 11: The radio of delay over months')
plt.grid(True,which="both",ls="-")
bar = ax.bar(index, pdf_top_20_carrier_volume['count'],
        bar_width, color='b',
        label='flight volume')
bar2 = ax2.bar(index + 1.5*bar_width, pdf_ratio_delay_carrier[1], bar_width
,
       align='center', color='r',
       label='Delay ratio')

lines, labels = ax.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc=0)

plt.tight_layout()
plt.show()
```



Comment

We can see in this figure that the carriers with a high flight volume always have a high delay ratio.

For instance, DL and US have the highest flight volume and also a high delay (>18%).

But other carriers also have a high delay ratio, even if they have a low flight volume (AS , HP and TW), so we can deduce that this carriers may have a bad service which causes delays.

4. Building a model of our data

Now that we have a good grasp on our data and its features, we will focus on how build a statistic model. Note that the features we can decide to use, to train our model, can be put in two groups:

- **Explicit features**: these are features that are present in the original data, or that can be built using additional data sources such as `weather` (for example querying a public API)
- **Implicit features**: these are the features that are inferred from other features such as

`is_weekend`, `is_holiday`, `season`, `in_winter`,...

In this notebook, we will focus on the following predictors: `year`, `month`, `day_of_month`, `day_of_week`, `scheduled_departure_time`, `scheduled_arrival_time`, `carrier`, `is_weekend`, `distance`, `src_airport`, `dest_airport`. Among them, `is_weekend` is an implicit feature. The rest are explicit features.

The target feature is `arrival_delay`.

Currently, MLLIB only supports building models from RDDs. It is important to read well the documentation and the MLLib API, to make sure to use the algorithms in an appropriate manner:

- MLLIB supports both categorical and numerical features. However, for each categorical feature, we have to indicate how many distinct values they can take
- Each training record must be a `LabelledPoint`. This data structure has 2 components: `label` and `predictor vector`. `label` is the value of target feature in the current record. `predictor vector` is a vector of values of type `Double`. As such, we need to map each value of each categorical feature to a number. In this project, we choose a naïve approach: map each value to a unique index.
- MLLIB uses a binning technique to find the split point (the predicate in each tree node). In particular, it divides the domain of numerical features into `maxBins` bins (32 by default). With categorical features, each distinct value fits in its own bin. **IMPORTANT:** MLLIB requires that no categorical feature have more than `maxBins` distinct values.
- We fill up the missing values in each **categorical** feature with its most common value. The missing values of a **numerical** feature are also replaced by the most common value (however, in some cases, a more sensible approach would be to use the median of this kind of feature).

4.1 Mapping values of each categorical feature to indices

Question 6

Among the selected features, `src_aiport`, `dest_airport`, `carrier` and `distance` have missing values. Besides, the first three of them are categorical features. That means, in order to use them as input features of MLLIB, the values of these features must be numerical. We can use a naïve approach: map each value of each feature to a unique index.

Question 6.1

Calculate the frequency of each source airport in the data and build a dictionary that maps each of them to a unique index. **Note:** we sort the airports by their frequency in descending order, so that we can easily take the most common airport(s) by taking the first element(s) in the result.

In [29]:

```
# select distinct source airports and map values to index
# sort the airport by their frequency descending
# so the most common aiport will be on the top
```

```
stat_src = (
    df
        .groupBy(df.src_airport)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

# extract the airport names from stat_src
src_airports = [item[0] for item in stat_src.collect()]

num_src_airports = len(src_airports)
src_airports_idx = range(0, num_src_airports)
map_src_airport_to_index = dict(zip(src_airports, src_airports_idx))

# test the dictionary
print(map_src_airport_to_index['ORD'])
print(map_src_airport_to_index['ATL'])
```

```
0
2
```

Question 6.2

> Calculate the frequency of each destination airport in the data and build a dictionary that maps each of them to a unique index.

In [30]:

```
# select distinct destination airports and map values to index
# sort the airport by their frequency descending
# so the most common aiport will be on the top
stat_dest = (
    df
        .groupBy(df.dest_airport)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

dest_airports =[item[0] for item in stat_dest.collect()]
num_dest_airports =len(dest_airports)
dest_airports_idx =range(0,num_dest_airports)
map_dest_airports_to_index = dict(zip(dest_airports, dest_airports_idx))

# test the dictionary
print(map_dest_airports_to_index['ORD'])
print(map_dest_airports_to_index['ATL'])
```

```
0
2
```

Question 6.3

> Calculate the frequency of each carrier in the data and build a dictionary that maps each of them to a unique index.

In [31]:

```
# select distinct carriers and map values to index
# sort carriers by their frequency descending
# so the most common aiport will be on the top
stat_carrier = (
    df
        .groupBy(df.carrier)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

carriers =[item[0] for item in stat_carrier.collect()]
num_carriers =len(carriers)
carriers_idx =range(0,num_carriers)
map_carriers_to_index =  dict(zip(carriers, carriers_idx))

# test the dictionary
print(map_carriers_to_index['DL'])
print(map_carriers_to_index['US'])
```

```
0
1
```

4.2 Calculating the most common value of each feature

We use a simple strategy for filling in the missing values: replacing them with the most common value of the corresponding feature.

IMPORTANT NOTE: features like ```month```, ```day_of_month```, etc... can be treated as numerical features in general. However, when it comes to build the model, it is much easier considering them as categorical features. In this case, to compute the most common value for such categorical features, we simply use the frequency of occurrence of each `label`, and chose the most frequent.

Question 7

In the previous question, when constructing the dictionary for categorical features, we also sort their statistical information in a such way that the most common value of each feature are placed on the top.

Note that, feature `is_weekend` has the most common value set to `0` (that is, `no` the day is not a weekend).

Question 7.1

Find the most common value of feature `month` in data.

In [32]:

```
the_most_common_month = (
    df
        .groupBy(df.month)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

print("The most common month:", the_most_common_month)
```

```
The most common month: 8
```

Question 7.2

> Find the most common value of features `day_of_month` and `day_of_week`.

In [33]:

```
the_most_common_day_of_month = (
    df
        .groupBy(df.day_of_month)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

the_most_common_day_of_week = (
    df
        .groupBy(df.day_of_week)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

print("The most common day of month:", the_most_common_day_of_month)
print("The most common day of week:", the_most_common_day_of_week)
```

```
The most common day of month: 11
The most common day of week: 3
```

Question 7.3

> Find the most common value of features `scheduled_departure_time` and `scheduled_arrival_time`.

In [34]:

```
the_most_common_s_departure_time = (
    df
        .groupBy(df.scheduled_departure_time)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

the_most_common_s_arrival_time = (
    df
```

```
        .groupBy(df.scheduled_arrival_time)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

print("The most common scheduled departure time:",
the_most_common_s_departure_time)
print("The most common scheduled arrival time:",
the_most_common_s_arrival_time)
```

```
The most common scheduled departure time: 700
The most common scheduled arrival time: 1915
```

Question 7.4

> Calculate the mean of distance in the data. This value will be used to fill in the missing values of feature `distance` later.

In [35]:

```
# calculate mean distance
stat_distance =(df
                    .agg(func.mean('distance').alias('mean_distance'))
                )

mean_distance = stat_distance.first()[0]
print("mean distance:", mean_distance)
```

```
mean distance: 670.7402911985982
```

Question 7.5

> Calculate the mean of arrival delay.

In [36]:

```
# calculate mean arrival delay
mean_arrival_delay = (df

.agg(func.mean('arrival_delay').alias('mean_arrival_delay'))
                ).first()[0]
print("mean arrival delay:", mean_arrival_delay)
```

```
mean arrival delay: 5.662489742613603
```

As known from section 3.4, there are 225 different origin airports and 225 different destination airports, more than the number of bins in default configuration. So, we must set `maxBins` >= 225.

4.3 Preparing training data and testing data

Recall, in this project we focus on decision trees. One way to think about our task is that we want to predict the unknown `arrival_delay` as a function combining several features, that is:

`arrival_delay = f(year, month, day_of_month, day_of_week, scheduled_departure_time, scheduled_arrival_time, carrier, src_airport, dest_airport, distance, is_weekend)`

When categorical features contain corrupt data (e.g., missing values), we proceed by replacing corrupt information with the most common value for the feature. For numerical features, in general, we use the same approach as for categorical features; in some cases, we repair corrupt data using the mean value of the distribution for numerical features (e.g., we found the mean for `delay` and `distance`, by answering questions above).

The original data is split randomly into two parts with ratios 70% for **training** and 30% for **testing**.

Question 8

o Replace the missing values of each feature in our data by the corresponding most common value or mean. o Divide data into two parts: 70% for **training** and 30% for **testing**

In [37]:

```
from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.regression import LabeledPoint


def is_valid(value):
    return value != "NA" and len(value) > 0

data = cleaned_data\
    .map(lambda line: line.split(','))\
    .map(lambda values:
        LabeledPoint(
            int(values[14]) if is_valid(values[14]) else mean_arrival_delay,
# arrival delay
            [
                int(values[0]), # year
                int(values[1]) if is_valid(values[1]) else
the_most_common_month, # month
                int(values[2]) if is_valid(values[2]) else
the_most_common_day_of_month, # day of month
                int(values[3]) if is_valid(values[3]) else
the_most_common_day_of_week, # day of week
                int(values[5]) if is_valid(values[5]) else
the_most_common_s_departure_time, # scheduled departure time
                int(values[7]) if is_valid(values[7]) else
the_most_common_s_arrival_time, # scheduled arrival time
                # if the value is valid, map it to the corresponding index
                # otherwise, use the most common value
                map_carriers_to_index[values[8]] if is_valid(values[8]) \
                    else map_carriers_to_index[carriers[0]], # carrier
                map_src_airport_to_index[values[16]] if is_valid(values[16])
else map_src_airport_to_index[src_airport[0]], # src_airport
                map_dest_airports_to_index[values[17]] if is_valid(values[17
]) else map_dest_airport_to_index[dest_airport[0]], # destination_airport
                int(values[18]) if is_valid(values[18]) else mean_distance,
```

```
# distance
                1 if is_valid(values[3]) and int(values[3]) >= 6 else 0, # i
s_weekend
            ]
        )
    )

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])

trainingData = trainingData.cache()
testData = testData.cache()
```

5.4 Building a decision tree model

Question 9

We can train a decision model by using function `DecisionTree.trainRegressor(, categoricalFeaturesInfo=, impurity=, maxDepth=, maxBins=)`.
Where,

- `training_data`: the data used for training
- `categorical_info`: a dictionary that maps the index of each categorical features to its number of distinct values
- `impurity_function`: the function that is used to calculate impurity of data in order to select the best split
- `max_depth`: the maximum depth of the tree
- `max_bins`: the maximum number of bins that the algorithm will divide on each feature.

Note that, `max_bins` cannot smaller than the number distinct values of every categorical features. Complete the code below to train a decision tree model.

In [38]:

```
# declare information of categorical features
# format:  feature_index : number_distinct_values
categorical_info = {6 : num_carriers, 7: num_src_airports, 8: num_dest_airpo
rts, 10: 2}

# Train a DecisionTree model.
model = DecisionTree.trainRegressor(trainingData,
                                    categoricalFeaturesInfo=categorical_inf
,
                                    impurity='variance', maxDepth=12, maxBi
=255)
```

5.5 Testing the decision tree model

Question 10

Question 10.1

We often use Mean Square Error as a metric to evaluate the quality of a tree model. Complete the code below to calculate the MSE of our trained model.

In [50]:

```
# Evaluate model on test instances and compute test error
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testMSE = labelsAndPredictions.map(
    lambda p: (p[0]-p[1])**2).mean()
print('Test Mean Squared Error = ' + str(testMSE))
```

```
Test Mean Squared Error = 493.2386894794151
```

Question 10.2

Comment the results you have obtained. Is the MSE value you get from a decision tree indicating that our statistical model is very good in predicting airplane delays? Use your own words to describe and interpret the value you obtained for the MSE.

Comment

With an MSE of 493 we have an average error of 21 minutes for every prediction ; knowing that the mean of the arrival delay is of 5.66 mins our model is really bad at predicting flight delays : maybe we have to gather more data or try a better algorithm.

5.6 Building random decision forest model (or random forest)

Next, we use MLLib to build a more powerful model: random forests. In what follows, use the same predictors defined and computed above to build a decision tree, but this time use them to build a random decision forest.

Question 11

Train a random decision forest model and evaluate its quality using MSE metric. Compare to decision tree model and comment the results. Similarly to question 10.2, comment with your own words the MSE value you have obtained.

In [40]:

```
from pyspark.mllib.tree import RandomForest, RandomForestModel

# Train a RandomForest model.
forest_model = RandomForest.trainRegressor(trainingData,
categoricalFeaturesInfo=categorical_info,
                                           numTrees=10, impurity='variance', maxDep
h=12, maxBins=255)

predictions = forest_model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testMSE = labelsAndPredictions.map(
    lambda p: np.square(p[0]-p[1])).sum()/testData.count()
print('Test Mean Squared Error = ' + str(testMSE))
```

```
Test Mean Squared Error = 480.791525947
```

Comment

Even with an advanced algorithm such as random forests we still get a high MSE, we can try finding additional predictors , or maybe gather more data.

5.7 Parameter tuning

In this lecture, we used `maxDepth=12`, `maxBins=255`, `numTrees=10`. Next, we are going to explore the meta-parameter space a little bit.

For more information about parameter tuning, please read the documentation of MLLIB

Question 12

Train the random forest model using different parameters, to understand their impact on the main performance metric we have used here, that is the MSE. For example, you can try a similar approach to that presented in the Notebook on recommender systems, that is using nested for loops.

NOTE: be careful when selecting parameters as some might imply very long training times, or eventually, the typical memory problems that affect Spark!

In [41]:

```
for r in [10,15,25]:
    for d in [12,14,16]:
        for b in [255,275]:
            forest_model = RandomForest.trainRegressor(trainingData,
```

```
categoricalFeaturesInfo=categorical_info,
                                    numTrees=r, impurity='variance', maxDept
=d, maxBins=b)
            predictions = forest_model.predict(testData.map(lambda x: x.feat
ures))
            labelsAndPredictions = testData.map(lambda lp: lp.label).zip(pre
dictions)
            testMSE = labelsAndPredictions.map(
                lambda p: np.square(p[0]-p[1])).mean()

            print('Test Mean Squared Error for numTrees = %f, maxDepth = %f
, maxBins = %f: %f' % (r,d,b,testMSE))
```

```
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 12.000000, max
Bins = 255.000000: 482.623636
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 12.000000, max
Bins = 275.000000: 482.187740
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 14.000000, max
Bins = 255.000000: 465.288706
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 14.000000, max
Bins = 275.000000: 461.682625
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 16.000000, max
Bins = 255.000000: 444.603141
Test Mean Squared Error for numTrees = 10.000000, maxDepth = 16.000000, max
Bins = 275.000000: 443.211929
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 12.000000, max
Bins = 255.000000: 480.146345
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 12.000000, max
Bins = 275.000000: 477.248812
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 14.000000, max
Bins = 255.000000: 459.380190
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 14.000000, max
Bins = 275.000000: 460.214362
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 16.000000, max
Bins = 255.000000: 442.220312
Test Mean Squared Error for numTrees = 15.000000, maxDepth = 16.000000, max
Bins = 275.000000: 442.907536
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 12.000000, max
Bins = 255.000000: 478.221502
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 12.000000, max
Bins = 275.000000: 477.677658
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 14.000000, max
Bins = 255.000000: 459.331416
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 14.000000, max
Bins = 275.000000: 458.994952
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 16.000000, max
Bins = 255.000000: 438.868106
Test Mean Squared Error for numTrees = 25.000000, maxDepth = 16.000000, max
Bins = 275.000000: 439.840202
```

6. Addition (bonus) questions

As you may have noticed, the performance of our statistical models is somehow questionable! What are we missing here? Why is that even using state-of-the-art approaches give poor results?

In what follows, we will try to address some of the limitations of the present Notebook, and provide additional data that might help.

6.1. Additional data

In the HDFS file system you have used for running the Notebook, you will notice that there are several other years available (in addition to 1994), which could be used to train a statistical model with more data. In the end, we're playing with "Big Data", hence one might think that feeding more training data to the algorithm should help!

6.2. Feature selection

You might think that the flight delays do not only depend on the source airport, destination airport, departure time, etc... as we assumed. They also depend on other features such as the weather, the origin country, the destination city,... To improve the prediction quality, we should consider these features too.

There are some other datasets that related to this use case:

- Airport IATA Codes to City names and Coordinates mapping: http://stat-computing.org/dataexpo/2009/airports.csv
- Carrier codes to Full name mapping: http://stat-computing.org/dataexpo/2009/carriers.csv
- Information about individual planes: http://stat-computing.org/dataexpo/2009/plane-data.csv
- Weather information: http://www.wunderground.com/weather/api/. You can subscribe for free to the developers' API and obtain (at a limited rate) historical weather information in many different formats. Also, to get an idea of the kind of information is available, you can use this link: http://www.wunderground.com/history/

Question 13

Using the data sources above, select additional feature and repeat the process of defining an appropriate training and test datasets, to evaluate the impact of new features on the performance of the model. Focus first on decision trees, then move to random forests.

> The important thing is to not stop questioning. Curiosity has its own reason for existence. (Albert Einstein)

Be active! Ask yourself other questions which help you explore more about this data and try to answer them. Make this notebook be a part of your CV!

In [64]:

```
input_path = "/datasets/airline/1994.csv"
raw_data = sc.textFile(input_path)
# extract the header
header = raw_data.first()

# replace invalid data with NULL and remove header
cleaned_data_main = (raw_data\
```

```
cleaned_data_main = (raw_data\
    # filter out the header
    .filter(lambda line: line != header)
    # replace the missing values with empty characters
    .map(lambda line: line.replace(',NA', ','))
)

for i in range(1995, 2005):
    input_path = "/datasets/airline/" + str(i) + ".csv"
    raw_data = sc.textFile(input_path)
    # extract the header
    header = raw_data.first()

    # replace invalid data with NULL and remove header
    cleaned_data = (raw_data\
        # filter out the header
        .filter(lambda line: line != header)
        # replace the missing values with empty characters
        .map(lambda line: line.replace(',NA', ','))
        )


    cleaned_data_main  = cleaned_data_main.union(cleaned_data)
cleaned_data = cleaned_data_main
print("number of rows after cleaning:", cleaned_data.count())
```

```
number of rows after cleaning: 62723910
```

In [65]:

```
# convert each line into a tuple of features (columns)
cleaned_data_to_columns = cleaned_data.map(lambda l: l.split(","))\
    .map(lambda cols:
         (
            int(cols[0])  if cols[0] else None,
            int(cols[1])  if cols[1] else None,
            int(cols[2])  if cols[2] else None,
            int(cols[3])  if cols[3] else None,
            int(cols[4])  if cols[4] else None,
            int(cols[5])  if cols[5] else None,
            int(cols[6])  if cols[6] else None,
            int(cols[7])  if cols[7] else None,
            cols[8]       if cols[8] else None,
            cols[9]       if cols[9] else None,
            cols[10]      if cols[10] else None,
            int(cols[11]) if cols[11] else None,
            int(cols[12]) if cols[12] else None,
            int(cols[13]) if cols[13] else None,
            int(cols[14]) if cols[14] else None,
            int(cols[15]) if cols[15] else None,
            cols[16]      if cols[16] else None,
            cols[17]      if cols[17] else None,
            int(cols[18]) if cols[18] else None,
            int(cols[19]) if cols[19] else None,
            int(cols[20]) if cols[20] else None,
            cols[21]      if cols[21] else None,
            cols[22]      if cols[22] else None,
            cols[23]      if cols[23] else None,
            int(cols[24]) if cols[24] else None,
            int(cols[25]) if cols[25] else None,
            int(cols[26]) if cols[26] else None,
```

```
            int(cols[27]) if cols[27] else None,
            int(cols[28]) if cols[28] else None
        ))
```

In [66]:

```
# create dataframe df
df = (sqlContext.createDataFrame(cleaned_data_to_columns,
airline_data_schema)
        .select(["year", "month", "day_of_month", "day_of_week", "scheduled_
departure_time", "scheduled_arrival_time", "arrival_delay"\
                , "distance", "src_airport", "dest_airport", "carrier"])
        .cache()
    )
```

In [67]:

```
# select distinct source airports and map values to index
# sort the airport by their frequency descending
# so the most common aiport will be on the top
stat_src = (
    df
        .groupBy(df.src_airport)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

stat_src.show()

# extract the airport names from stat_src
src_airports = [item[0] for item in stat_src.collect()]

num_src_airports = len(src_airports)
src_airports_idx = range(0, num_src_airports)
map_src_airport_to_index = dict(zip(src_airports, src_airports_idx))

# test the dictionary
print(map_src_airport_to_index['ORD'])
print(map_src_airport_to_index['ATL'])
```

```
+-----------+-------+
|src_airport|  count|
+-----------+-------+
|        ORD|3466352|
|        DFW|3031577|
|        ATL|2999154|
|        LAX|2157540|
|        PHX|1863078|
|        STL|1660289|
|        DTW|1631556|
|        MSP|1542883|
|        DEN|1498002|
|        IAH|1494327|
|        SFO|1406441|
|        LAS|1398099|
|        EWR|1369029|
|        CLT|1347071|
|        BOS|1194591|
|        PHL|1194585|
|        LGA|1135396|
```

```
|        LGA|1135396|
|        PIT|1129113|
|        SEA|1085548|
|        SLC| 987244|
+-----------+-------+
only showing top 20 rows

0
2
```

In [68]:

```
# select distinct destination airports and map values to index
# sort the airport by their frequency descending
# so the most common aiport will be on the top
stat_dest = (
    df
        .groupBy(df.dest_airport)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

stat_dest.show()

dest_airports = [item[0] for item in stat_dest.collect()]
num_dest_airports = len(dest_airports)
map_dest_airports_to_index = dict(zip(dest_airports, range(0,
num_dest_airports)))

# test the dictionarycreate
print(map_src_airport_to_index['ORD'])
print(map_src_airport_to_index['ATL'])
```

```
+------------+-------+
|dest_airport|  count|
+------------+-------+
|         ORD|3472703|
|         DFW|3035699|
|         ATL|2996700|
|         LAX|2157610|
|         PHX|1864336|
|         STL|1661608|
|         DTW|1634145|
|         MSP|1544292|
|         DEN|1500911|
|         IAH|1494711|
|         SFO|1405584|
|         LAS|1398390|
|         EWR|1369404|
|         CLT|1347948|
|         PHL|1194941|
|         BOS|1193537|
|         LGA|1134793|
|         PIT|1129776|
|         SEA|1085037|
|         SLC| 988004|
+------------+-------+
only showing top 20 rows

0
2
```

In [69]:

```
# select distinct carriers and map values to index
# sort carriers by their frequency descending
# so the most common aiport will be on the top
stat_carrier = (
    df
        .groupBy(df.carrier)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )

stat_carrier.show()

carriers = [item[0] for item in stat_carrier.collect()]
num_carriers = len(carriers)

map_carriers_to_index = dict(zip(carriers, range(0, num_carriers)))
```

```
+-------+-------+
|carrier|  count|
+-------+-------+
|     WN|9254106|
|     DL|9218204|
|     AA|7839291|
|     UA|7534691|
|     US|7275397|
|     NW|5753535|
|     CO|4233939|
|     HP|2204939|
|     TW|2148625|
|     MQ|1841588|
|     AS|1651356|
|     OO| 860249|
|     XE| 695789|
|     DH| 556555|
|     EV| 552573|
|     OH| 373346|
|     FL| 307258|
|     B6| 157018|
|     TZ| 144709|
|     AQ|  64834|
+-------+-------+
only showing top 20 rows
```

In [70]:

```
the_most_common_month = (
    df
        .groupBy(df.month)
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    ).first()[0]

map_int_into_month = { 1:"Jan", 2:"Feb", 3:"Mar", 4:"Apr", 5:"May", 6:"Jun",
7:"Jul", 8:"Aug", 9:"Sep", 10:"Oct", 11:"Nov", 12:"Dec" }

print("The most common month:", map_int_into_month[the_most_common_month])
```

```
The most common month: Aug
```

In [71]:

```
the_most_common_day_of_month = (
    df
    .groupBy(df.day_of_month)
    .agg(func.count('*').alias('count'))
    .orderBy(desc('count'))
).first()[0]

the_most_common_day_of_week = (
    df
    .groupBy(df.day_of_week)
    .agg(func.count('*')
         .alias('count'))
    .orderBy(desc('count'))
).first()[0]

map_int_into_day = { 1:"Mon", 2:"Tue", 3:"Wed", 4:"Thu", 5:"Fri", 6:"Sat", 7
:"Sun" }

print("The most common day of month:", the_most_common_day_of_month)
print("The most common day of week:",
map_int_into_day[the_most_common_day_of_week])
```

```
The most common day of month: 19
The most common day of week: Mon
```

In [72]:

```
the_most_common_s_departure_time = (
    df
    .groupBy(df.scheduled_departure_time)
    .agg(func.count('*').alias('count'))
    .orderBy(desc('count'))
).first()[0]

the_most_common_s_arrival_time = (
    df
    .groupBy(df.scheduled_arrival_time)
    .agg(func.count('*').alias('count'))
    .orderBy(desc('count'))
).first()[0]

def int_to_hour(time):
    "Int to hour format"
    return str(time)[:-2] + ':' + str(time)[-2:]

print("The most common scheduled departure time:", int_to_hour(the_most_com
mon_s_departure_time))
print("The most common scheduled arrival time:", int_to_hour(the_most_commo
n_s_arrival_time))
```

```
The most common scheduled departure time: 7:00
The most common scheduled arrival time: :0
```

In [73]:

```
# calculate mean distance
mean_distance = (
```

```
    df.agg(func.mean('distance'))
).first()[0]

print("mean distance:", mean_distance)
```

```
mean distance: 725.005449432039
```

In [74]:

```
# calculate mean arrival delay
mean_arrival_delay = (
    df.agg(func.mean('arrival_delay'))
    ).first()[0]

print("mean arrival delay:", mean_arrival_delay)
```

```
mean arrival delay: 6.759091745816796
```

In [75]:

```
def is_valid(value):
    return value != "NA" and len(value) > 0

data = cleaned_data\
    .map(lambda line: line.split(','))\
    .map(lambda values:
        LabeledPoint(
            int(values[14]) if is_valid(values[14]) else mean_arrival_delay,
# arrival delay
            [
                int(values[0]), # year
                int(values[1]) if is_valid(values[1]) else
the_most_common_month, # month
                int(values[2]) if is_valid(values[2]) else
the_most_common_day_of_month, # day of month
                int(values[3]) if is_valid(values[3]) else
the_most_common_day_of_week, # day of week
                int(values[5]) if is_valid(values[5]) else
the_most_common_s_departure_time, # scheduled departure time
                int(values[7]) if is_valid(values[7]) else
the_most_common_s_arrival_time, # scheduled arrival time
                # if the value is valid, map it to the corresponding index
                # otherwise, use the most common value
                map_carriers_to_index[values[8]] if is_valid(values[8]) \
                    else map_carriers_to_index[carriers[0]], # carrier
                map_src_airport_to_index[values[16]] if is_valid(values[16])
else map_src_airport_to_index[src_airport[0]], # src_airport
                map_dest_airports_to_index[values[17]] if is_valid(values[17
]) else map_dest_airport_to_index[dest_airport[0]], # destination_airport
                int(values[18]) if is_valid(values[18]) else mean_distance,
# distance
                1 if is_valid(values[3]) and int(values[3]) >= 6 else 0, # i
s_weekend
            ]
        )
    )
```

In [76]:

```
# Split the data into training and test sets (30% held out for testing)
```

```
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])

trainingData = trainingData.cache()
testData = testData.cache()
```

In [77]:

```
num_src_airport = df.select("src_airport").distinct().count()
num_dest_airport = df.select("dest_airport").distinct().count()
print("number of origin airports ", num_src_airport)
print("number of destination airports ", num_dest_airport)
```

```
number of origin airports  301
number of destination airports  304
```

In [79]:

```
categorical_info = {6 : num_carriers, 7: num_src_airports, 8: num_dest_airpo
rts, 10: 2}

# Train a DecisionTree model.
model = DecisionTree.trainRegressor(trainingData,
                                    categoricalFeaturesInfo=categorical_inf
,
                                    impurity='variance', maxDepth=12, maxBi
=304)
```

In [80]:

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

testMSE = labelsAndPredictions.map(
    lambda p: (p[0] - p[1])**2).reduce(lambda x, y: x + y) / labelsAndPredict
ions.count()
print('Test Mean Squared Error = ' + str(testMSE))
```

```
Test Mean Squared Error = 870.9282871157013
```

Comment

The MSE is almost double what we had before when we used data from 10 years , this means that the features we chose are not are not representative for predicting the delay , we should work on finding better features.

Feature selection : From the datasets proposed we can select new features that can be helpfull for our model:

--> From Airport IATA Codes to City names and Coordinates mapping , we can infer a categorical feature about the flights : wether it is intra-state , intra-country , inter-country,intercontinental.

--> From Weather information we can infer the influence of weather state on delays : rainy,sunny,windy... and label our data accordingly.

--> We can use the plane models to make categories of planes depending on the delays they make.

make?

Summary

In this lecture, we've had an overview about Decision Trees, Random Forests and how to use them. We also insisted on a simple methodology to adopt when dealing with a Data Science problem. The main take home messages should be:

- Feature selection is a difficult, delicate and important task. In this project, the student was heavily guided. However, we invite to exercise with additional features, for example external ones related to weather conditions.
- Parameter tuning requires a deep understanding of the algorithm used to build a statistical model. In general, to reduce computational cost, several techniques introduce parameters that, if tuned properly, can lead to tremendous time savings.