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*.

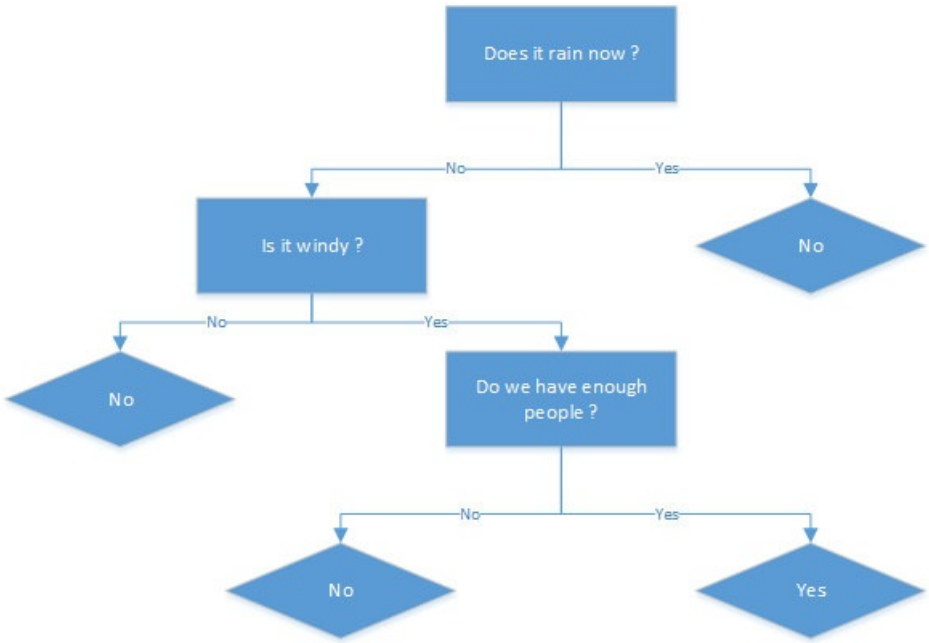
1.2 Decision Trees & Random Forests

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 (http://stat-computing.org/dataexpo/2009/the-data.html), 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 line : line!=header)
        # replace the missing values with empty characters
        .map(lambda l: l.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',
            'arrival_delay', 'distance',
            'src_airport',  'dest_airport', 'carrier','tail_number'])\
    .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*".
  - 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
```

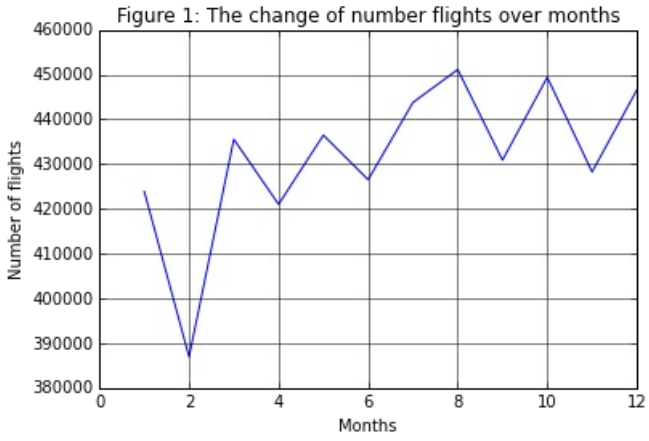
Question 4: Flight volume statistics

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'])

#statistic_month.show()

#statistic_day_of_week.show()
pdf = pd.DataFrame(data=statistic_month.take(12))

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()
```



We can clearly notice that february has the lowest number of flights while August is the one with the highest number of flights. This is due to the " Low season / High season " effect.

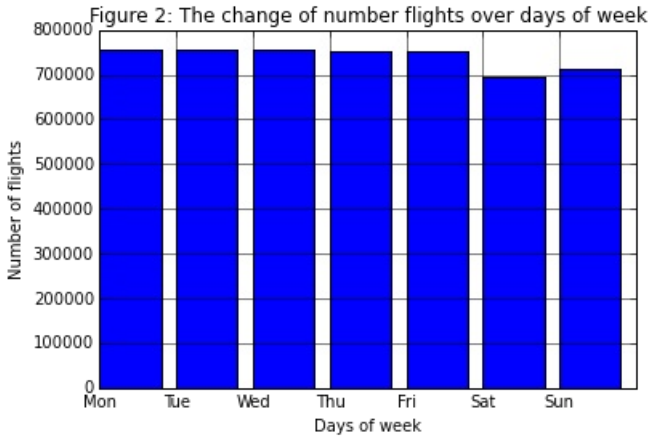
Question 4.2:

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()
#statistic_day_of_week.show()
pdf = pd.DataFrame(data=statistic_day_of_week.take(7))
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])
print(pdf[0])
# plot bar chart
plt.bar(pdf[0],pdf[1])

plt.xticks(pdf[0], day_of_week_label)
plt.show()
```

```
0    1
1    6
2    3
3    5
4    4
5    7
6    2
Name: 0, dtype: int64
```



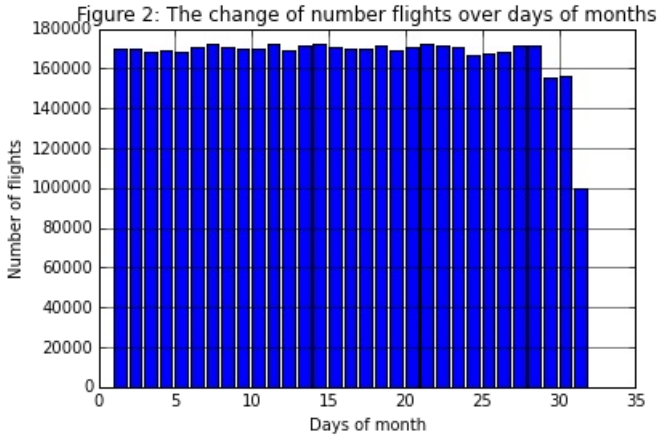
The number of flights during the weekend is less important than the number of flights during the rest of week. This may be due to the busniss trips that usually take place during the week.

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]:

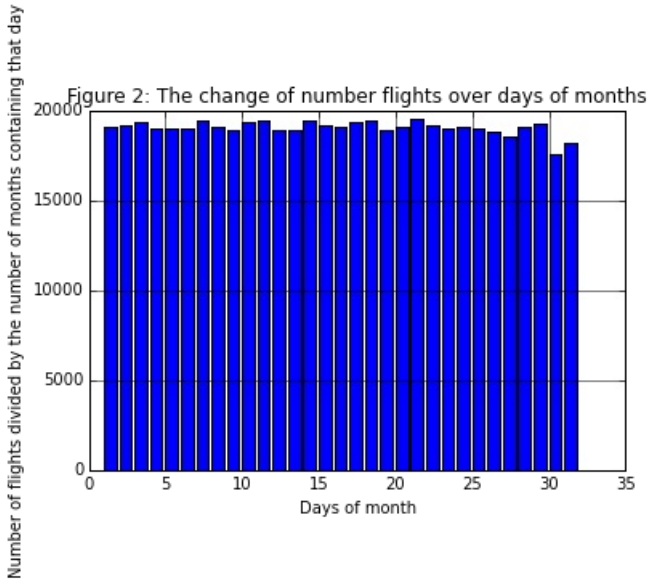
```
#days of month

statistic_daysmonth = df.groupby('day_of_month').count().sort(['day_of_month'])
#statistic_month.show()
#statistic_day_of_week.show()
pdf = pd.DataFrame(data=statistic_daysmonth.take(31))
plt.xlabel("Days of month")
plt.ylabel("Number of flights")
plt.title('Figure 2: The change of number flights over days of months')
plt.grid(True,which="both",ls="-")
plt.bar(pdf[0],pdf[1])

plt.show()
```



We can notice that the Day 31th is the one with the least number of flights. This is normal because only one month over two have a 31th day. So there are only 7 of it per year instead of 12 for the other days. Let's try to take into account this fact : We compute the mean of flights over the 7 days 31th of the year, and the mean of the flights over each other day of the year. (for instance there are 7 months ending with a day 31st, 11 months having a day 30th, 12 having a day 15th, ...)

In [59]:

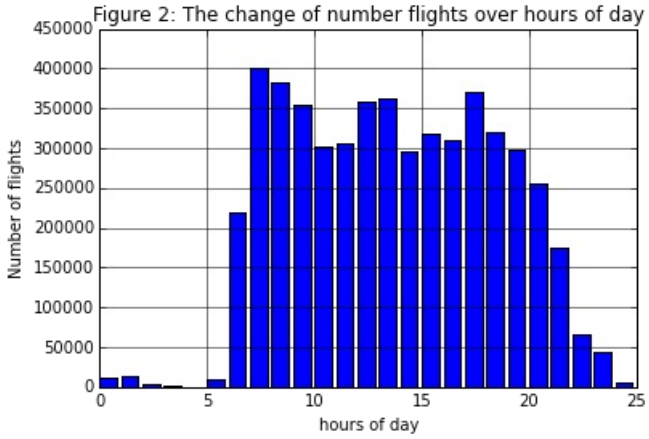
```
df2 = df.withColumn('numberofmonths', when(df['day_of_month'] != 31, 12).otherwise(7))


statistic_daysmonth = df2.groupby('day_of_month').agg((func.count('*')/func.mean('numberofmonths')).alias('c
ountt')).sort(['day_of_month'])



pdf = pd.DataFrame(data=statistic_daysmonth.take(31))

plt.xlabel("Days of month")
plt.ylabel("Number of flights divided by the number of months containing that day ")
plt.title('Figure 2: The change of number flights over days of months')
plt.grid(True,which="both",ls="-")
plt.bar(pdf[0],pdf[1])

plt.show()
```



Notice that we should have devided the flights of the day 30th by 11, but we didn't want to make the code too complicated for something that doesn't have a great interest. If this was done, then the bar of the day 30th would has been slightly different.

In [14]:

```
df_with_hourofday = df.withColumn('hour', round(df.scheduled_departure_time/100, 0))



statistic_hour_of_day = df_with_hourofday.groupby('hour').count().sort(['hour'])
pdf = pd.DataFrame(data=statistic_hour_of_day.take(25))
plt.xlabel("hours of day")
plt.ylabel("Number of flights")
plt.title('Figure 2: The change of number flights over hours of day')
plt.grid(True,which="both",ls="-")
plt.bar(pdf[0],pdf[1])
plt.show()
```



We can notice that the number of flights between midnight and 5 AM are very few compared to the rest of day. On the other hand, many flights are scheduled around 8 AM and 5 PM.

Question 4.4

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

In [15]:

```
# consider outbound flights
stat_src = (df.groupBy(df.src_airport).agg(func.count('*').alias('count1')))
stat_src.show(5)

# 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_src[0]==stat_dest[0] , how='full')

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

```
+-----------+------+
|src_airport|count1|
+-----------+------+
|        BGM|  1432|
|        PSE|    65|
|        DLG|   308|
|        MSY| 48055|
|        GEG|  8392|
+-----------+------+
only showing top 5 rows

+-----------+------+
|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 [16]:

```
stat_carrier = (df.groupBy('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

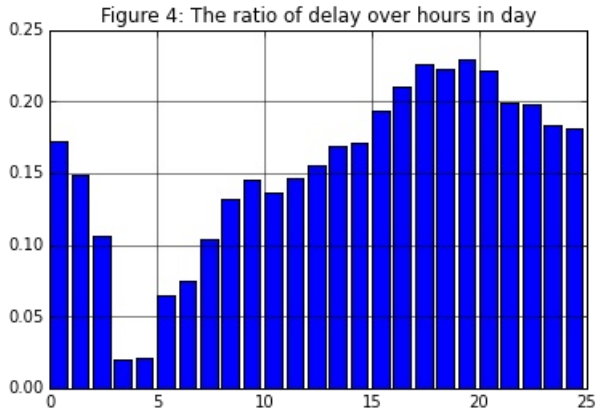
What is the percentage of delayed flights for different hours of the day? Plot a bar chart and interpret the figure. **Remember** a flight is considered as delayed if it's actual arrival time is more than 15 minutes late than the scheduled arrival time.

In [17]:

```
# 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
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']))

pdf_delay_ratio_per_hour = pd.DataFrame(data=delay_ratio_per_hour.take(25))
```

In [18]:

```
# plot a bar chart



plt.grid(True,which="both",ls="-")
plt.bar(pdf_delay_ratio_per_hour[0],pdf_delay_ratio_per_hour[1])


plt.title('Figure 4: The ratio of delay over hours in day')
plt.show()
```



In [19]:

```
statistic_delay_hour = delay_per_hour.groupBy(['year','hour'])

# 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']))

pdf_delay_ratio_per_hour = pd.DataFrame(data=delay_ratio_per_hour.take(25))
```

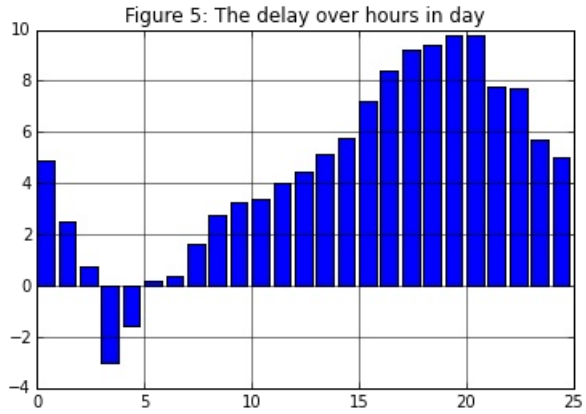
The ratio of delay is high around 8 PM, and low around 4 AM. This can be simply related to the fact that around 4 AM there are very few flights compared to around 8 PM (see Figure 3 question 4.3).

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 [20]:

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

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

pdf_mean_delay_per_hour = pd.DataFrame(data=mean_delay_per_hour.take(25))



plt.grid(True,which="both",ls="-")
plt.bar(pdf_mean_delay_per_hour[0],pdf_mean_delay_per_hour[1])

plt.title('Figure 5: The delay over hours in day')
plt.show()
```



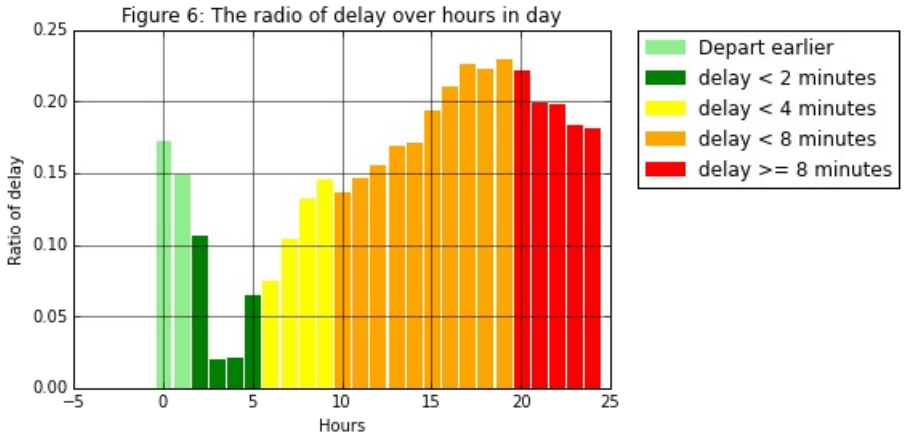
The flights of the year 1994 from 3AM to 4AM have often taken off earlier than what was scheduled. We can notice also that the morning flights have less delay than the other flights. This is due to the fact that there are only few flights scheduled early in the morning.(cf. figure 2)

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 [21]:

```
#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], align='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.

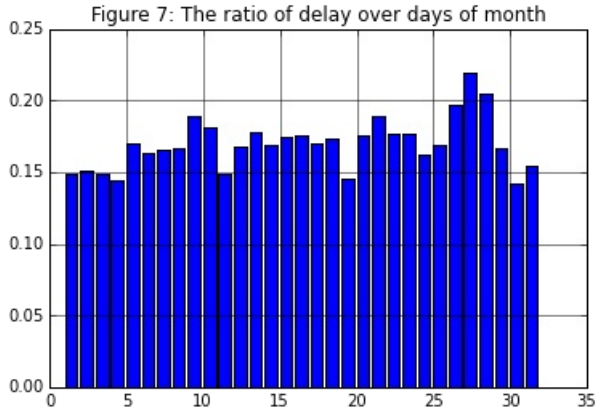
This example shows us that the way representing results are also important.

Question 5.3

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

In [60]:

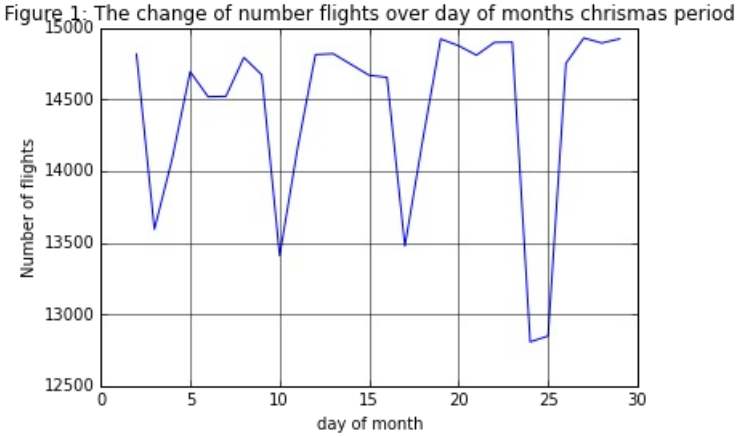
```
####### The changes of delay ratio over days of month ########
# calculate the delay ratio in each day of month
statistic_day_of_month = (
    df_with_delay
        .groupBy('day_of_month')
        .agg((func.sum('is_delay')/func.count('*')))
        # order by day_of_month
        .orderBy('day_of_month'))


# collect data and plot
pdf_day_of_month = pd.DataFrame(data=statistic_day_of_month.take(31))

plt.grid(True,which="both",ls="-")
plt.bar(pdf_day_of_month[0],pdf_day_of_month[1])
plt.title('Figure 7: The ratio of delay over days of month')
plt.show()
```



it is noticeable that the most delayed flights take place between the day 26th and the 28th of a month.

In [23]:

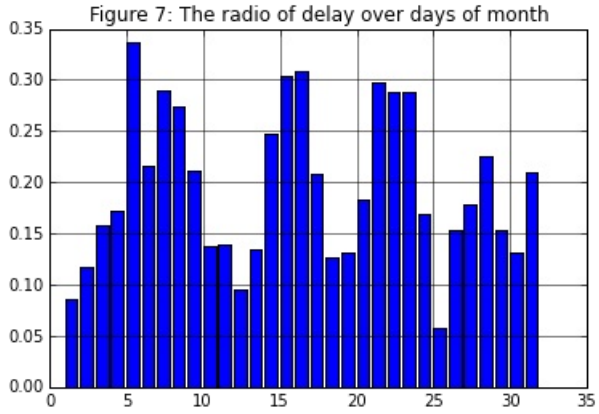
```
chrismasstat=df.filter('month==12').filter('day_of_month<30').filter('day_of_month>1').groupBy(['month','day
_of_month']).count().select(['day_of_month','count']).sort('day_of_month')

pdf = pd.DataFrame(data=chrismasstat.take(30))

plt.xlabel("day of month")
plt.ylabel("Number of flights")
plt.title('Figure 1: The change of number flights over day of months chrismas period ')
plt.grid(True,which="both",ls="-")
plt.plot(pdf[0],pdf[1])
plt.show()
```



During the christmas day (24-25) , the number of flights decreases significantly, because obviously people prefer to celebrate rather than traveling.

In [24]:

```
statistic_day_of_month = (
    df_with_delay
        .filter('month==12')
        .groupBy('day_of_month')
        .agg((func.sum('is_delay')/func.count('*')))
        # order by day_of_month
        .orderBy('day_of_month'))


# collect data and plot
pdf_day_of_month = pd.DataFrame(data=statistic_day_of_month.take(31))

plt.grid(True,which="both",ls="-")
plt.bar(pdf_day_of_month[0],pdf_day_of_month[1])
plt.title('Figure 7: The radio of delay over days of month')
plt.show()
```

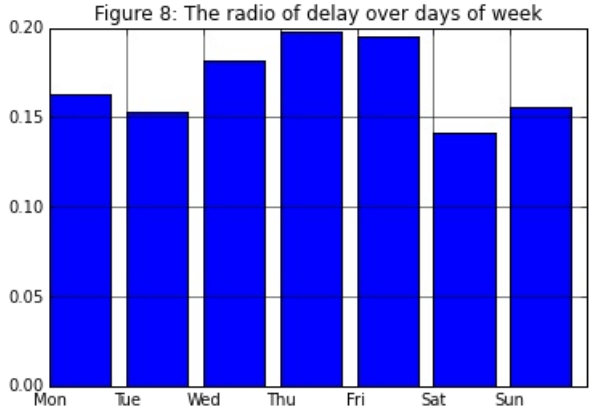


Question 5.4

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

In [25]:

```
####### 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('day_of_week')
        .agg((func.sum('is_delay')/func.count('*')))
        # order by day_of_month
        .orderBy('day_of_week'))

# collect data and plot
pdf_day_of_week = pd.DataFrame(data=statistic_day_of_week.take(7))
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.grid(True,which="both",ls="-")
plt.bar(pdf_day_of_week[0],pdf_day_of_week[1])

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()
```



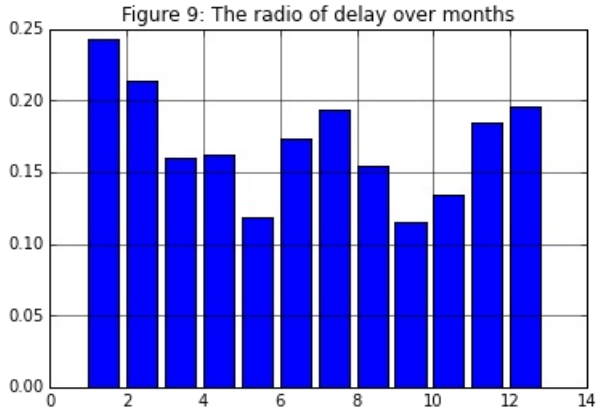
The days where the most delayed flights take place are thursday and Friday.

Question 5.5

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

In [26]:

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

# collect data and plot
pdf_month = pd.DataFrame(data=statistic_month.take(12))

plt.grid(True,which="both",ls="-")
plt.bar(pdf_month[0],pdf_month[1])

plt.title('Figure 9: The radio of delay over months')
plt.show()
```



January is the month with the highest ratio of delay over months. This is due to the fact that this month is in the high seazon, which means a lot of flights, and obviously more delays too.

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?)

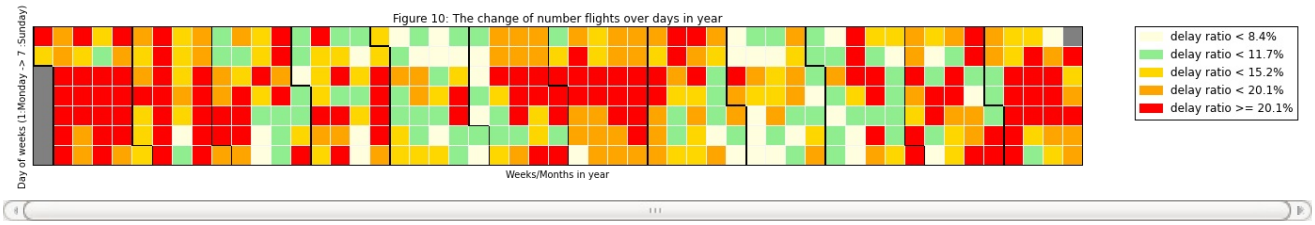
Putting things together, we can have a global picture of the whole year!

In [27]:

```
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 [28]:

```
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).timetuple()
    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.

The delimited Repartitions represent the months. Each column represents a week, and each square is a given day. The months that contain a lot of red are those who are very bad to travel, for instance january. While the months that contain a lot of white and green are those that are good for traveling, as the delays are very low. For example, May and september.

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.

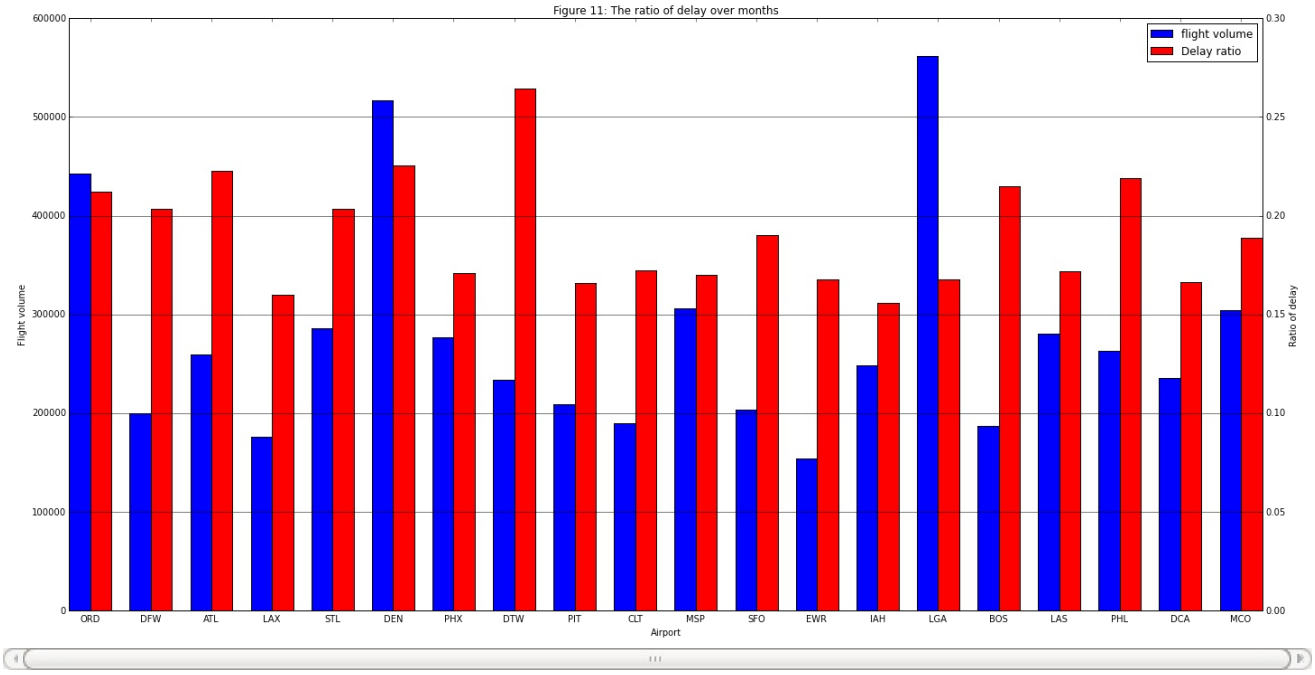
In [29]:

```
####### 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.mean('is_delay')))
        # sort by name of airport
        .orderBy(['src_airport'])
    )
statistic_ratio_delay_airport.show(20)
```

```
+-----------+-------------------+
|src_airport|      avg(is_delay)|
+-----------+-------------------+
|        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 [30]:

```
# collect data and plot
pdf_ratio_delay_airport = pd.DataFrame(data=statistic_ratio_delay_airport.collect())
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_airport')
#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()
```



We can notice that the Flight volume and the delay ratio are not linked to each other. The delays may be principaly due to the way each aireport handles flights.

Question 5.8

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

In [31]:

```
top_20_carriers = [item[0] for item in stat_carrier.take(20)]
print(top_20_carriers)
```

```
['DL', 'US', 'AA', 'UA', 'WN', 'CO', 'NW', 'TW', 'HP', 'AS']
```

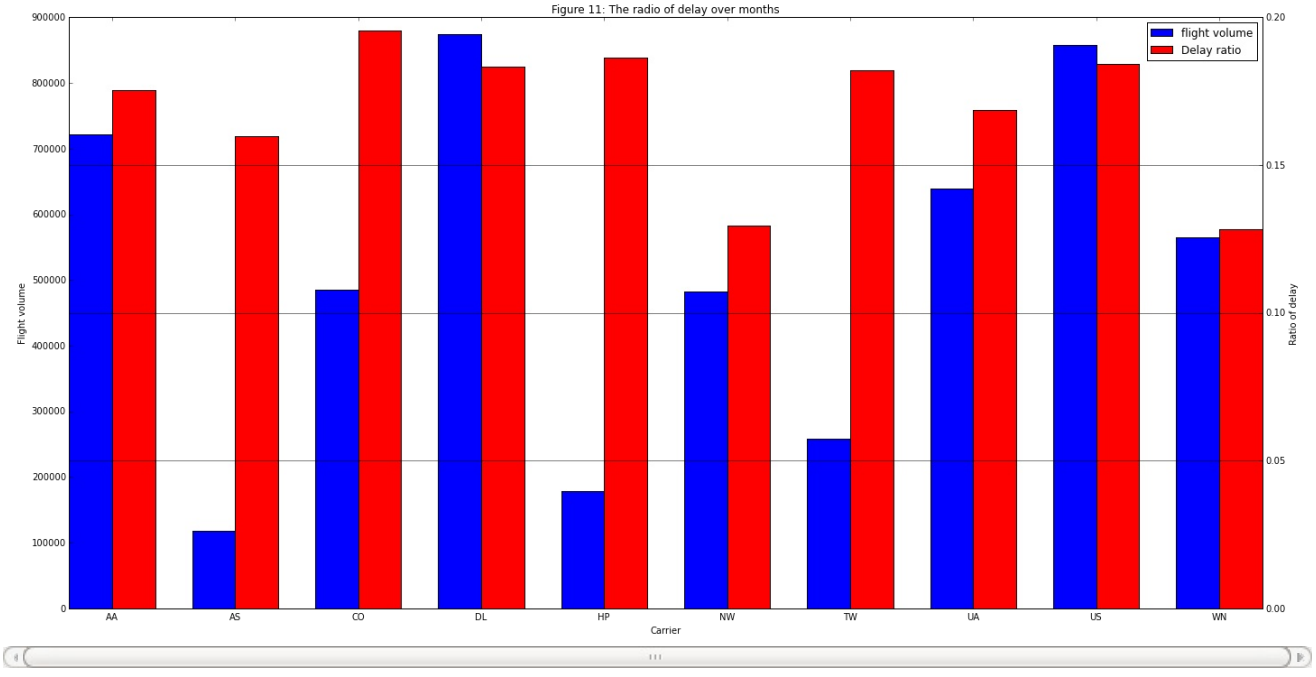
In [32]:

```
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.mean('is_delay')))
        # sort by name of airport
        .orderBy(['carrier'])

    )
statistic_ratio_delay_carrier.show(20)
```

```
+-------+------------------+
|carrier|      avg(is_delay)|
+-------+------------------+
|     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 [33]:

```
# collect data and plot
pdf_ratio_delay_carrier = pd.DataFrame(data=statistic_ratio_delay_carrier.collect())
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='carrier')
#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 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()
```



This figure confirms somehow the comment we wrote above : There is no special link between flight volume and delay ratio, we think that the delays are more correlated to each aireport flights managment, ...

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 [34]:

```
stat_src = (
    df
        .groupBy('src_airport')
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )
stat_src.show(20)
```

```
+-----------+------+
|src_airport| count|
+-----------+------+
|        ORD|277741|
|        DFW|256274|
|        ATL|221486|
|        LAX|153494|
|        STL|151609|
|        DEN|141427|
|        PHX|139581|
|        DTW|136837|
|        PIT|130977|
|        CLT|129391|
|        MSP|123222|
|        SFO|118233|
|        EWR|116768|
|        IAH|104182|
|        LGA|101864|
|        BOS|100365|
|        LAS| 94972|
|        PHL| 93585|
|        DCA| 88445|
|        MCO| 77120|
+-----------+------+
only showing top 20 rows
```

In [35]:

```
# 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('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 [36]:

```
# 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('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 [37]:

```
# 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('carrier')
        .agg(func.count('*').alias('count'))
        .orderBy(desc('count'))
    )
carrier_ = [item[0] for item in stat_carrier.collect()]
num_carrier = len(carrier_)
carrier_idx = range(0,num_carrier )
map_carriers_to_index = dict(zip(carrier_, carrier_idx))

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 [38]:

```
the_most_common_month = (
    df
        .groupBy('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 [39]:

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

the_most_common_day_of_week = (
    df
        .groupBy('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 [40]:

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

the_most_common_s_arrival_time = (
    df
        .groupBy('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 [41]:

```
# calculate mean distance
mean_distance = df.agg(func.mean('distance')).first()[0]
print("mean distance:", mean_distance)
```

```
mean distance: 670.7402911985982
```

Question 7.5

Calculate the mean of arrival delay.

In [42]:

```
# 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: 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 [43]:

```
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 dep
arture time
                int(values[7]) if is_valid(values[7]) else the_most_common_s_arrival_time, # scheduled arriv
al 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[carrier_[
0]], # carrier
                map_src_airport_to_index[values[16]] if is_valid(values[16]) else map_src_airport_to_index[s
rc_airports[0]], # src_airport
                map_dest_airports_to_index[values[17]] if is_valid(values[17]) else map_dest_airports_to_ind
ex[dest_airports[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, # is_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 [44]:

```
# declare information of categorical features
# format:  feature_index : number_distinct_values

categorical_info = {6 : num_carrier, 7:num_src_airports , 8: num_dest_airports, 10: 2 }

# Train a DecisionTree model.
model = DecisionTree.trainRegressor(trainingData,
                                    categoricalFeaturesInfo=categorical_info,
                                    impurity='variance', maxDepth=12, maxBins=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 [45]:

```
# 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 = 491.54054052917263
```

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.

The MSE of 491.54 corresponds nearly to a mean of 22 minutes delay. So we assume that our model is not good in predicting airplane delays since the mean arrival delay is equal to 5.66 (question 7.5)

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 [46]:

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

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

predictionss = forest_model.predict(testData.map(lambda x: x.features))
labelsAndPredictionss = testData.map(lambda lp: lp.label).zip(predictionss)
testMSE = labelsAndPredictionss.map(
    lambda p: (p[0]-p[1])**2).mean()
print('Test Mean Squared Error = ' + str(testMSE))
```

```
Test Mean Squared Error = 480.75697544699125
```

the mean square error is still too high, yet we used an advanced algorithm which is the random decision forest. We should maybe take more information from the data,or use more elements for the predictions.

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 (http://spark.apache.org/docs/latest/mllib-decision-tree.html#tunable-parameters)

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 [47]:

```
categorical_info = {6 : num_carrier, 7:num_src_airports , 8: num_dest_airports, 10: 2 }


evaluations = []

for maxDepth in [10, 15]:
    for maxBins in [255, 300]:
        for numTrees in [8,16 ]:
            print("Train model with maxDepth=%d maxBins=%f numTrees=%f" % (maxDepth, maxBins, numTrees))

            forest_model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo=categorical_inf
o,numTrees=numTrees, impurity='variance', maxDepth=maxDepth, maxBins=maxBins)
            predictionss = forest_model.predict(testData.map(lambda x: x.features))
            labelsAndPredictionss = testData.map(lambda lp: lp.label).zip(predictionss)
            testMSE = labelsAndPredictionss.map(lambda p: (p[0]-p[1])**2).mean()


            evaluations.append(((maxDepth, maxBins, numTrees), testMSE))



evaluations.sort( key=lambda x : -x[1] )

evalDataFrame = pd.DataFrame(data=evaluations)
print(evalDataFrame)
```

```
Train model with maxDepth=10 maxBins=255.000000 numTrees=8.000000
Train model with maxDepth=10 maxBins=255.000000 numTrees=16.000000
Train model with maxDepth=10 maxBins=300.000000 numTrees=8.000000
Train model with maxDepth=10 maxBins=300.000000 numTrees=16.000000
Train model with maxDepth=15 maxBins=255.000000 numTrees=8.000000
Train model with maxDepth=15 maxBins=255.000000 numTrees=16.000000
Train model with maxDepth=15 maxBins=300.000000 numTrees=8.000000
Train model with maxDepth=15 maxBins=300.000000 numTrees=16.000000
               0           1
0   (10, 255, 8)  499.477410
1  (10, 255, 16)  497.731392
2   (10, 300, 8)  497.520224
3  (10, 300, 16)  497.351037
4   (15, 255, 8)  460.977258
5   (15, 300, 8)  457.013846
6  (15, 300, 16)  452.239081
7  (15, 255, 16)  451.732387
```

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 (http://stat-computing.org/dataexpo/2009/airports.csv)
- Carrier codes to Full name mapping: http://stat-computing.org/dataexpo/2009/carriers.csv (http://stat-computing.org/dataexpo/2009/carriers.csv)
- Information about individual planes: http://stat-computing.org/dataexpo/2009/plane-data.csv (http://stat-computing.org/dataexpo/2009/plane-data.csv)
- Weather information: http://www.wunderground.com/weather/api/ (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/ (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 [48]:

```
input_path2 = "plane-data.csv"
raw_data2 = sc.textFile(input_path2)
```

In [49]:

```
header2 = raw_data2.first()


# replace invalid data with NULL and remove header

cleaned_data2 = (raw_data2\
        # filter out the header
        .filter(lambda line : line!=header2)
        # replace the missing values with empty characters
        .map(lambda l: l.replace(',NA', ','))
        .filter(lambda line : len(line) > 8)
        )

print("number of rows after cleaning:",cleaned_data2.count() )

cleaned_data2.take(5)
```

```
number of rows after cleaning: 4480
```

Out[49]:

```
['N10156,Corporation,EMBRAER,02/13/2004,EMB-145XR,Valid,Fixed Wing Multi-Engine,Turbo-Fan,2004'
,
 'N102UW,Corporation,AIRBUS INDUSTRIE,05/26/1999,A320-214,Valid,Fixed Wing Multi-Engine,Turbo-F
an,1998',
 'N10323,Corporation,BOEING,07/01/1997,737-3TO,Valid,Fixed Wing Multi-Engine,Turbo-Jet,1986',
 'N103US,Corporation,AIRBUS INDUSTRIE,06/18/1999,A320-214,Valid,Fixed Wing Multi-Engine,Turbo-F
an,1999',
 'N104UA,Corporation,BOEING,01/26/1998,747-422,Valid,Fixed Wing Multi-Engine,Turbo-Fan,1998']
```

In [50]:

```
sqlContext = SQLContext(sc)
airline_data_schema2 = StructType([ \
    #StructField( name, dataType, nullable)
    StructField("tail_number",                    StringType(), True), \
    StructField("type",                   StringType(), True), \
    StructField("manufacturer",            StringType(), True), \
    StructField("issue_date",             StringType(), True), \
    StructField("model",           StringType(), True), \
    StructField("status", StringType(), True), \
    StructField("aircraft_type",             StringType(), True), \
    StructField("engine_type",   StringType(), True), \
    StructField("year",                 StringType(),  True), \
])
```

In [51]:

```
cleaned_data_to_columns2 = cleaned_data2.map(lambda l: l.split(","))\
    .map(lambda cols:
         (
            cols[0]  if cols[0] else None,
            cols[1]  if cols[1] else None,
            cols[2]  if cols[2] else None,
            cols[3]  if cols[3] else None,
            cols[4]  if cols[4] else None,
            cols[5]  if cols[5] else None,
            cols[6]  if cols[6] else None,
            cols[7]  if cols[7] else None,
            cols[8]  if cols[8] else None
            ))
```

In [52]:

```
df2 =  sqlContext.createDataFrame(cleaned_data_to_columns2, airline_data_schema2)
```

In [53]:

```
df2.show(5)
```

```
+-----------+-----------+---------------+----------+---------+------+--------------------+----
-------+----+
|tail_number|       type|    manufacturer|issue_date|    model|status|       aircraft_type|engi
ne_type|year|
+-----------+-----------+---------------+----------+---------+------+--------------------+----
-------+----+
|     N10156|Corporation|         EMBRAER|02/13/2004|EMB-145XR| Valid|Fixed Wing Multi-...|  Tu
rbo-Fan|2004|
|     N102UW|Corporation|AIRBUS INDUSTRIE|05/26/1999| A320-214| Valid|Fixed Wing Multi-...|  Tu
rbo-Fan|1998|
|     N10323|Corporation|          BOEING|07/01/1997|  737-3TO| Valid|Fixed Wing Multi-...|  Tu
rbo-Jet|1986|
|     N103US|Corporation|AIRBUS INDUSTRIE|06/18/1999| A320-214| Valid|Fixed Wing Multi-...|  Tu
rbo-Fan|1999|
|     N104UA|Corporation|          BOEING|01/26/1998|  747-422| Valid|Fixed Wing Multi-...|  Tu
rbo-Fan|1998|
+-----------+-----------+---------------+----------+---------+------+--------------------+----
-------+----+
only showing top 5 rows
```

In [54]:

```
df22=df2.select(['tail_number','issue_date'])
```

In [55]:

```
input_path = "/datasets/airline/2008.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 l: l.replace(',NA', ','))
        )

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


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)\
])

# 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
         ))

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','tail_number'])\
    .cache()
```

```
number of rows after cleaning: 7009728
```

In [56]:

```
df.show(10)

df22.show(5)
```

```
+----+-----+------------+-----------+----------------------+---------------------+----------
---+--------+-----------+------------+-------+-----------+
|year|month|day_of_month|day_of_week|scheduled_departure_time|scheduled_arrival_time|arrival_de
lay|distance|src_airport|dest_airport|carrier|tail_number|
+----+-----+------------+-----------+----------------------+---------------------+----------
---+--------+-----------+------------+-------+-----------+
|2008|    1|           3|          4|                  1955|                 2225|
-14|     810|        IAD|        TPA|     WN|     N712SW|
|2008|    1|           3|          4|                   735|                 1000|
  2|     810|        IAD|        TPA|     WN|     N772SW|
|2008|    1|           3|          4|                   620|                  750|
 14|     515|        IND|        BWI|     WN|     N428WN|
|2008|    1|           3|          4|                   930|                 1100|
 -6|     515|        IND|        BWI|     WN|     N612SW|
|2008|    1|           3|          4|                  1755|                 1925|
 34|     515|        IND|        BWI|     WN|     N464WN|
|2008|    1|           3|          4|                  1915|                 2110|
 11|     688|        IND|        JAX|     WN|     N726SW|
|2008|    1|           3|          4|                  1830|                 1940|
 57|    1591|        IND|        LAS|     WN|     N763SW|
|2008|    1|           3|          4|                  1040|                 1150|
-18|    1591|        IND|        LAS|     WN|     N428WN|
|2008|    1|           3|          4|                   615|                  650|
  2|     451|        IND|        MCI|     WN|     N689SW|
|2008|    1|           3|          4|                  1620|                 1655|
-16|     451|        IND|        MCI|     WN|     N648SW|
+----+-----+------------+-----------+----------------------+---------------------+----------
---+--------+-----------+------------+-------+-----------+
only showing top 10 rows

+-----------+----------+
|tail_number|issue_date|
+-----------+----------+
|     N10156|02/13/2004|
|     N102UW|05/26/1999|
|     N10323|07/01/1997|
|     N103US|06/18/1999|
|     N104UA|01/26/1998|
+-----------+----------+
only showing top 5 rows
```

In [57]:

```
dfend=df22.join(df,on='tail_number',how ='left')
```

In [58]:

```
dfend.show(5)
```

```
+-----------+----------+----+-----+------------+-----------+----------------------+----------
------------+-------------+--------+-----------+------------+-------+
|tail_number|issue_date|year|month|day_of_month|day_of_week|scheduled_departure_time|scheduled_
arrival_time|arrival_delay|distance|src_airport|dest_airport|carrier|
+-----------+----------+----+-----+------------+-----------+----------------------+----------
------------+-------------+--------+-----------+------------+-------+
|     N102UW|05/26/1999|2008|    1|           1|          2|                  1640|
       1909|           23|    2125|        CLT|         LAX|     US|
|     N102UW|05/26/1999|2008|    1|           1|          2|                   725|
       1456|          -19|    2125|        LAX|         CLT|     US|
|     N102UW|05/26/1999|2008|    1|           2|          3|                  2225|
       2321|           39|     156|        CLT|         MYR|     US|
|     N102UW|05/26/1999|2008|    1|           2|          3|                  1630|
       1826|           33|     590|        CLT|         PBI|     US|
|     N102UW|05/26/1999|2008|    1|           2|          3|                   725|
       1456|            8|    2125|        LAX|         CLT|     US|
+-----------+----------+----+-----+------------+-----------+----------------------+----------
------------+-------------+--------+-----------+------------+-------+
only showing top 5 rows
```

Comment

Next , we will load the 2008 dataset "/datasets/airline/2008.csv", and join it with what we need in "plane-data.csv" using ".join(df,on='tail_number',how ='left')" . Indeed we will try to use the same features as before adding to them the age of the plane . Notice that we kept the last model but for the year 2008 with the same old features . We only made a join with the plane data in order to add the age of the plane to it because we guess that the age of the plane can impact the delays since the plane needs more time for maintenance for example

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.

In [ ]: