Skalering til Big Data - Jan Pedersbaek

June 10, 2018

1 Handin, Master i IT

2 Jan Pedersbaek

2.1 Skalering til Big Data (MIT), Sunday, June 10th, 2018

2.1.1 1) Explain your choice of processing framework briefly.

It've chosen to do the exercises in PySPark. Python libraries have nice plotting features and jupyter notebook is my tool of choice, when it comes to combining ad-hoc analysis and reporting. However, there are some considerations to do, when using the python API for Spark. * Parsing objects between Spark and Python (serializing) is a quite expensive task, so this should be done with care. Most often, the reason for doing so regardless of the performance-hit is, that plotting is required. In these cases however, I will seek to aggregate data on Spark dataframes before moving to Python native formats. Generally, I will persist Spark dataframes to memory, when these are to be used multiple times. Spark transformations are executed in a lazy manner, meaning that nothing happens before an action is called on the dataframe. Until then, the dataframe only exists logically in the form of "data linage". However, when a dataframe is persisted, it will be kept in memory, and in cases where this should be accessed repeatedly - as in many ML algorithms (and certanly ingrid-search operations), this will speed up the process significantly.

```
In [1]: #import pyspark as spark
        #import findspark
        #import pyspark
        from pyspark.sql import SparkSession
        from pyspark.sql.functions import *
        import pandas as pd
```

```
In [2]: import matplotlib.pyplot as plt
```

For this project, Spark is set up to run locally, which means without a real cluster of physical machines. However parallelism is still very much in effect and therefor considerations on how many executers that should be set up still apply. The setup for this project is as follows: * The machine is cloud-based (AWS), which means that scaling ressources up and down is easy - this has come in handy a couple of times * Final setup is a AWS EC2 t2.2xLARGE running Ubuntu * Configuration is 8vCPU and 32Gb of ram * Spark is configured with 30Gb of memory allocated to the driver-process (leaving 2Gb for the OS). Running locally, executers will run in the same context, and Spark will automaticall allocate a certain fraction of the total allocated memory to executers. * Spark is initiated with “Master Local[*]” meaning, that the number of executers will match the number of CPUt’s on the system, in this case 8.

```
In [3]: # https://stackoverflow.com/questions/26562033/how-to-set-apache-spark-executor-memory
        sc._conf.get('spark.driver.memory')
```

```
Out[3]: '30g'
```

This project is done on data on US domestic flights in the year 2008. The dataset has already been downloaded manually, and below is one approach to defining a Spark Dataframe using a StructType. This gives the posibillity of manually defining the schema that should be used when reading in the flat file.

```
In [4]: from pyspark.sql.types import StructType, StructField, IntegerType, StringType
        schema = StructType([
            StructField("Year", IntegerType(), True),
            StructField("Month", IntegerType(), True),
            StructField("DayofMonth", IntegerType(), True),
            StructField("DayOfWeek", IntegerType(), True),
            StructField("DepTime", IntegerType(), True),
            StructField("CRSDepTime", IntegerType(), True),
            StructField("ArrTime", IntegerType(), True),
            StructField("CRSArrTime", IntegerType(), True),
            StructField("UniqueCarrier", StringType(), True),
            StructField("FlightNum", IntegerType(), True),
            StructField("TailNum", StringType(), True),
            StructField("ActualElapsedTime", IntegerType(), True),
            StructField("CRSElapsedTime", IntegerType(), True),
            StructField("AirTime", IntegerType(), True),
            StructField("ArrDelay", IntegerType(), True),
            StructField("DepDelay", IntegerType(), True),
            StructField("Origin", StringType(), True),
            StructField("Dest", StringType(), True),
            StructField("Distance", IntegerType(), True),
            StructField("TaxiIn", IntegerType(), True),
            StructField("TaxiOut", IntegerType(), True),
            StructField("Cancelled", IntegerType(), True),
            StructField("CancellationCode", StringType(), True),
            StructField("Diverted", IntegerType(), True),
            StructField("CarrierDelay", IntegerType(), True),
            StructField("WeatherDelay", IntegerType(), True),
            StructField("NASDelay", IntegerType(), True),
            StructField("SecurityDelay", IntegerType(), True),
            StructField("LateAircraftDelay", IntegerType(), True)])
```

Flights data is loaded (or rather, defined because no load has actually happened yet because of the lazy nature of Spark)

```
In [5]: flights = spark.read.csv("./data/2008.csv",header=True,schema=schema, nullValue='NA')
```

Another option is to let Sparn "infer" the schema, meaning that it will guess from the first number of rows. Examples of this is seen below, where additional dataframes are defined

```
In [6]: airports = spark.read.csv("./data/airports.csv",\
                                  header=True,\
                                  inferSchema=True,\
                                  nullValue='NA')

        airlines = spark.read.csv("./data/carriers.csv",\
                                  header=True,\
                                  inferSchema=True,\
                                  nullValue='NA')
```

2.1.2 2. How many flights were there from JFK to LAX?

Having defined the flights data, lets answer the first couple of questions in the assignment :Finding the number of flights from JFK to LAX

```
In [7]: flights.where((col('Origin') == 'JFK') & (col('Dest') == 'LAX')).\
        count()

Out[7]: 8078
```

2.1.3 3. What was the sum and average of all arrival delays for all delayed flights?

Finding the sum and average of all arrival delays for all delayed flights Averages could be found using "Describe", but to include sum, we will use select. Furthermore, we only want to calculate the average delay for those that were actually delayed.

```
In [8]: flights.where(col("ArrDelay")>0).select(avg('ArrDelay'),\
                                                sum('ArrDelay'))\
        .show()

+-----------------+-------------+
|    avg(ArrDelay)|sum(ArrDelay)|
+-----------------+-------------+
|32.170706265203876|     95852748|
+-----------------+-------------+
```

2.1.4 4. What was the average departure delay for each state?

To answer this, we need the airport data from airports.csv, that we already defined above.

```
In [9]: airports.show(2)

+----+--------------------+-----------+-----+-------+-----------+------------+
|iata|             airport|       city|state|country|        lat|        long|
+----+--------------------+-----------+-----+-------+-----------+------------+
```

```
| 00M|           Thigpen |Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal| Livingston|   TX|    USA|30.68586111|-95.01792778|
+----+--------------------+-----------+-----+-------+-----------+------------+
only showing top 2 rows
```

Now, lets join the dataframes, group the result on states (from the airport data) and calculate the average departure-delay.To illustrate the "agg" function used with a map (instead of just a simple "sum"), wet'll add the average arrival-delays aswell.

```
In [10]: # Wet'll do the join and persist, since we will use this dataframe later on aswell
         # Broadcast airports if possible
         flightsWithAirports = flights.join(airports, flights.Origin == airports.iata)

In [11]: delays = flightsWithAirports.\
             groupBy(airports.state).\
                 agg({"DepDelay": "avg", "ArrDelay": "avg"}).\
                     select(col("state").alias("state"), \
                            col("avg(DepDelay)").alias("avgDepDelay"), \
                            col("avg(ArrDelay)").alias("avgArrDelay")).\
                   sort(desc("avgDepDelay"))

         delays.show()
```

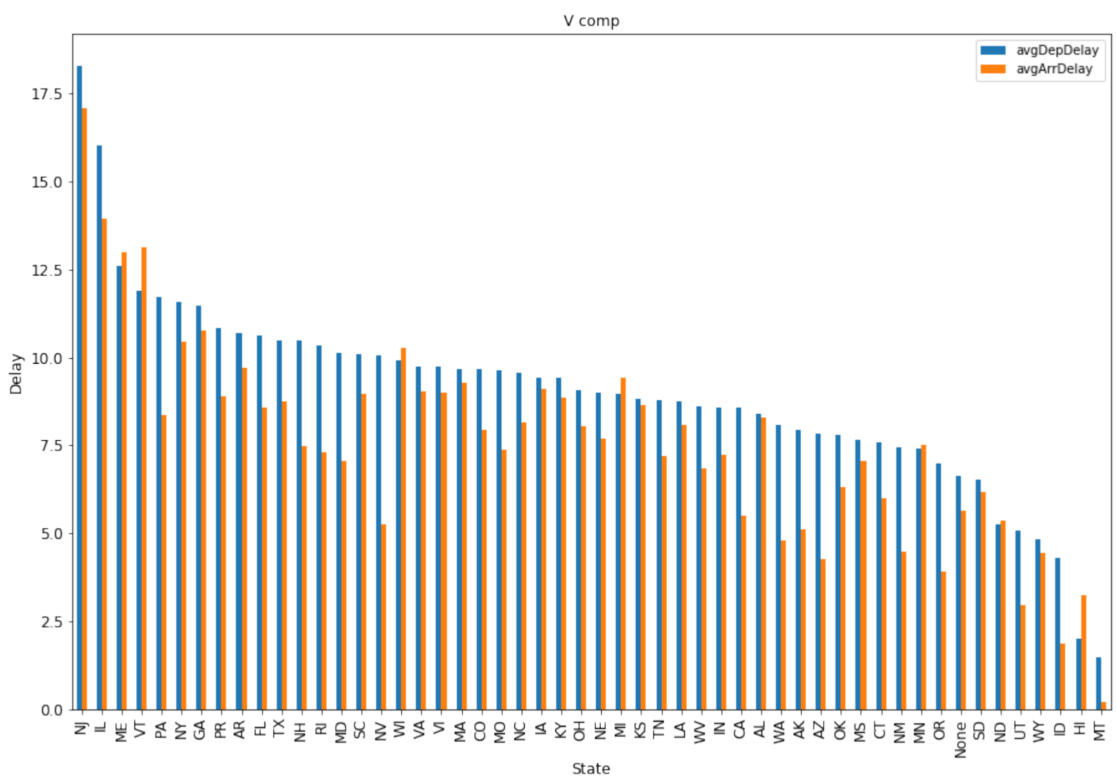
```
+-----+------------------+------------------+
|state|       avgDepDelay|       avgArrDelay|
+-----+------------------+------------------+
|   NJ| 18.28530315230682|17.073619219183303|
|   IL|16.037485162920703|13.927999295439097|
|   ME| 12.60202895487689|12.972307692307693|
|   VT|11.906676449009538| 13.11985294117647|
|   PA|11.706605875610164| 8.359997157696183|
|   NY|11.581353889575762|10.433212329260538|
|   GA| 11.47578943937115|10.746965986839188|
|   PR|10.823683322079676| 8.884239061374899|
|   AR|10.697886119257086| 9.709514325111076|
|   FL|10.617784856557332| 8.554335060599021|
|   TX|10.484268832380778| 8.741412350982355|
|   NH|10.483407140123559| 7.463268777088934|
|   RI|10.345095558668053| 7.284535521603119|
|   MD|10.136788700696506|7.0616724670931115|
|   SC|10.073743016759776| 8.942515845928815|
|   NV|10.047854928293972| 5.234664517182271|
|   WI| 9.898691052537206|10.273451327433628|
|   VA| 9.741461461852408| 9.015987468487651|
|   VI| 9.727703703703703|  9.00453446191052|
|   MA| 9.677755692715417| 9.280603542532255|
+-----+------------------+------------------+
```

```
only showing top 20 rows
```

Quite difficult to get a sense of this result, so lets visualize it. PySpark does not have plotting capabillities per se, so wet'll convert the Spark-dataframe to a pandas dataframe (requires installing python Pandas and MatPlotLib libraries). Pandas has several easy-to-use plotting features, and sorting the data by descending departure delay will give us a visual sense of the correlation btw departure delay and arrival delay (state-wise). Serializing the data and pasing them to Python should not be that big a deal, as data is already pretty aggregated (state-level).

```
In [12]: pdDelays = delays.toPandas()
```

```
In [13]: %matplotlib inline
         states=pdDelays['state'].tolist()
         #states
         ax = pdDelays[['avgDepDelay','avgArrDelay']].\
             plot(kind='bar',\
                  title ="V comp",\
                  figsize=(15, 10),\
                  legend=True,\
                  fontsize=12)

         ax.set_xticklabels(states)
         ax.set_xlabel("State", fontsize=12)
         ax.set_ylabel("Delay", fontsize=12)
         plt.show()
```



2.1.5 5. Which airline performed the worst seen from a customer perspective ?

Analysing airlines, lets first load the carriers.csv file, that contains carrier-names instead of just codes. As seen above, "show()" on a Spark dataframe is not that pretty, so again Pandas to the rescure, as there is not that much data to pass to Python. Still, lets limit to 5 rows, just to get a sense of the dataframe:

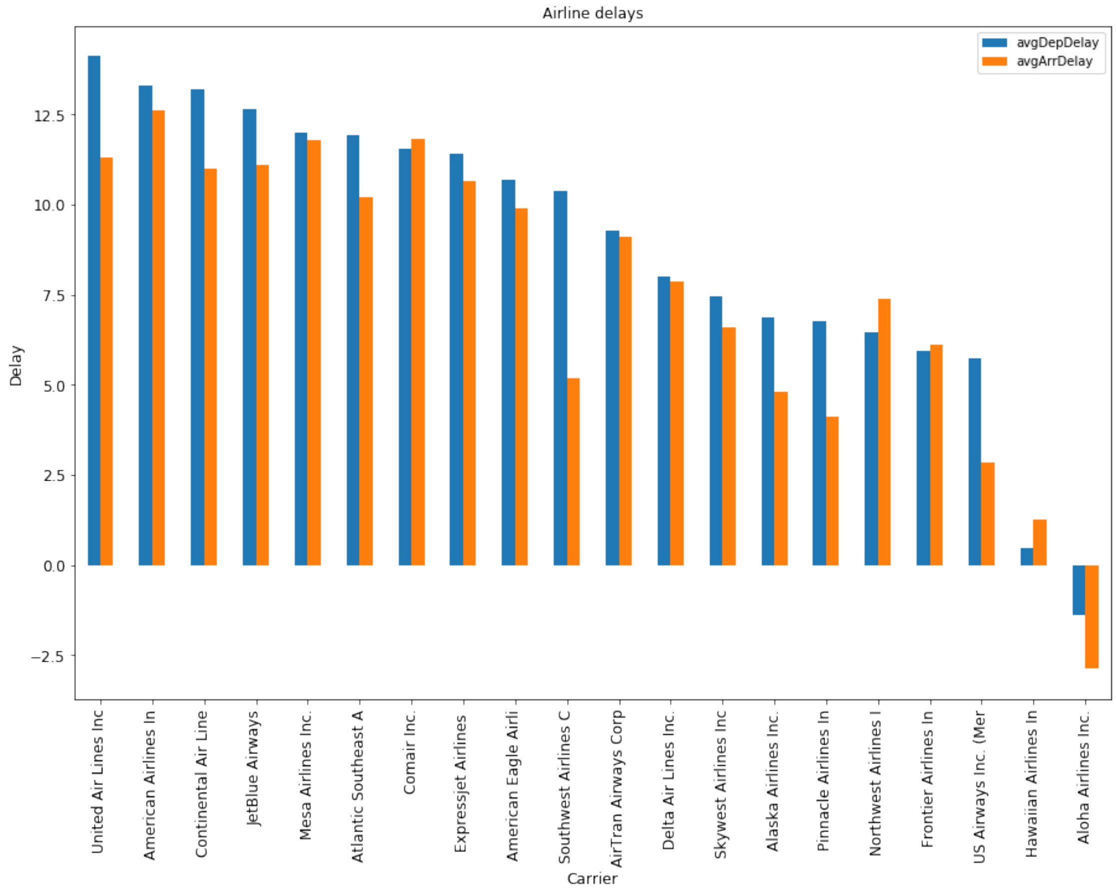
```
In [14]: airlines.limit(5).toPandas()
```

```
Out[14]:   Code                      Description
        0  02Q                   Titan Airways
        1  04Q              Tradewind Aviation
        2  05Q             Comlux Aviation, AG
        3  06Q  Master Top Linhas Aereas Ltd.
        4  07Q             Flair Airlines Ltd.
```

Now, lets take a look at each airline to determine, which performed the worst, seen from a customer perspective. At first, we'll plot the same delay-data as above, but with an airline focus instead of airport/state focus. Again, we'll aggregate data on Spark before parsing to Python (Pandas) for plotting

```
In [15]: from pyspark.sql.functions import substring
         carrierDelays = flights.join(airlines,flights.UniqueCarrier==airlines.Code,\
                                      "left_outer").\
         groupBy(airlines.Description).\
         agg({"DepDelay": "avg", "ArrDelay": "avg"}).\
         select(col("Description").alias("Description"), \
                col("avg(DepDelay)").alias("avgDepDelay"),\
                col("avg(ArrDelay)").alias("avgArrDelay")).\
         sort(desc("avgDepDelay")).\
         withColumn("Desc", substring("Description", 1, 20)).\
         toPandas()

         carriers=carrierDelays['Desc'].tolist()

         ax = carrierDelays[['avgDepDelay','avgArrDelay']].\
             plot(kind='bar',\
                  title ="Airline delays",\
                  figsize=(15, 10),\
                  legend=True,\
                  fontsize=12)

         ax.set_xticklabels(carriers)
         plt.xticks(rotation=90)
         ax.set_xlabel("Carrier", fontsize=12)
         ax.set_ylabel("Delay", fontsize=12)
```

```
plt.show()
#print(carrierDelays.to_string(index=False))
```



Now, delays is not all that matters from a customert's point of view, so wet'll compute a wide range of statistics to describe the airline performance. Still, all these descriptive measures do not give us a single "best airline" metric, so lets also choose a couple of them and create a general performance-measure:

- depOnTimePct
- arrOnTimePct
- completedFlightsPct

These are all percentages (eg. values between 0 and 1) describing positive feautures, where 1 is "perfect" and 0 is "worst". If we multiply these measures for each airline, again 1 would describe "perfect performance" and 0 would describe "worst possible performance". Lets rank the airlines according to this airline performance measure:

```
In [75]: # 1) flight-level feature engeneering, eg creating new features
         # 2) Grouping by carrier
         # 3) Aggregating metrics pr. carrier
         # 4) Calculation percentage metrics on carrier level
```

```
carrierPerformanceTable = flights.\
    select(flights.UniqueCarrier, \
           flights.DepDelay, \
           when(flights.DepDelay > 0,1).otherwise(0).alias("IsDepDelayed"),\
           when(flights.DepDelay > 0,0).otherwise(1).alias("IsDepOnTime"),\
           when(flights.ArrDelay > 0,1).otherwise(0).alias("IsArrDelayed"),\
           when(flights.ArrDelay > 0,0).otherwise(1).alias("IsArrOnTime"),\
           when(flights.Cancelled== 0,1).otherwise(0).alias("Completed"),\
           flights.DepDelay,
           flights.ArrDelay,
           flights.Cancelled
          ).\
    groupBy(flights.UniqueCarrier). \
    agg(sum("DepDelay").alias("DepDelay"), \
        max("DepDelay").alias("maxDepDelay"), \
        sum("ArrDelay").alias("ArrDelay"), \
        max("ArrDelay").alias("maxArrDelay"), \
        sum("IsDepDelayed").alias("isDepDelayed"), \
        sum("IsDepOnTime").alias("isDepOnTime"), \
        sum("IsArrDelayed").alias("isArrDelayed"), \
        sum("IsArrOnTime").alias("isArrOnTime"), \
        sum("Cancelled").alias("isCancelled"),\
        sum("Completed").alias("isCompleted"),\
        count(lit(1)).alias("numberOfFlights") \
       ). \
    select(col("UniqueCarrier"),\
           ((col("IsCompleted") / col("numberOfFlights"))*\
            (col("IsDepOnTime") / col("numberOfFlights"))*\
            (col("IsArrOnTime") / col("numberOfFlights"))).\
               alias("performanceMeasure"),\
           round(col("IsDepOnTime") / col("numberOfFlights")*100,2).\
               alias("depOnTimePct"),\
           round(col("IsArrOnTime") / col("numberOfFlights")*100,2).\
               alias("arrOnTimePct"),\
           round(col("IsDepDelayed") / col("numberOfFlights")*100,2).\
               alias("depDelayedPct"),\
           round(col("IsArrDelayed") / col("numberOfFlights")*100,2).\
               alias("arrDelayedPct"),\
           round(col("DepDelay") / col("isDepDelayed"),2).\
               alias("AvgDepDelayWhenDelayed"),\
           round(col("ArrDelay") / col("isArrDelayed"),2).\
               alias("AvgArrDelayWhenDelayed"),\
           round(col("MaxArrDelay"),2).\
               alias("MaxArrDelay"),\
           round(col("MaxDepDelay"),2).\
               alias("MaxDepDelay"),\
           round(col("isCancelled"),2).\
```

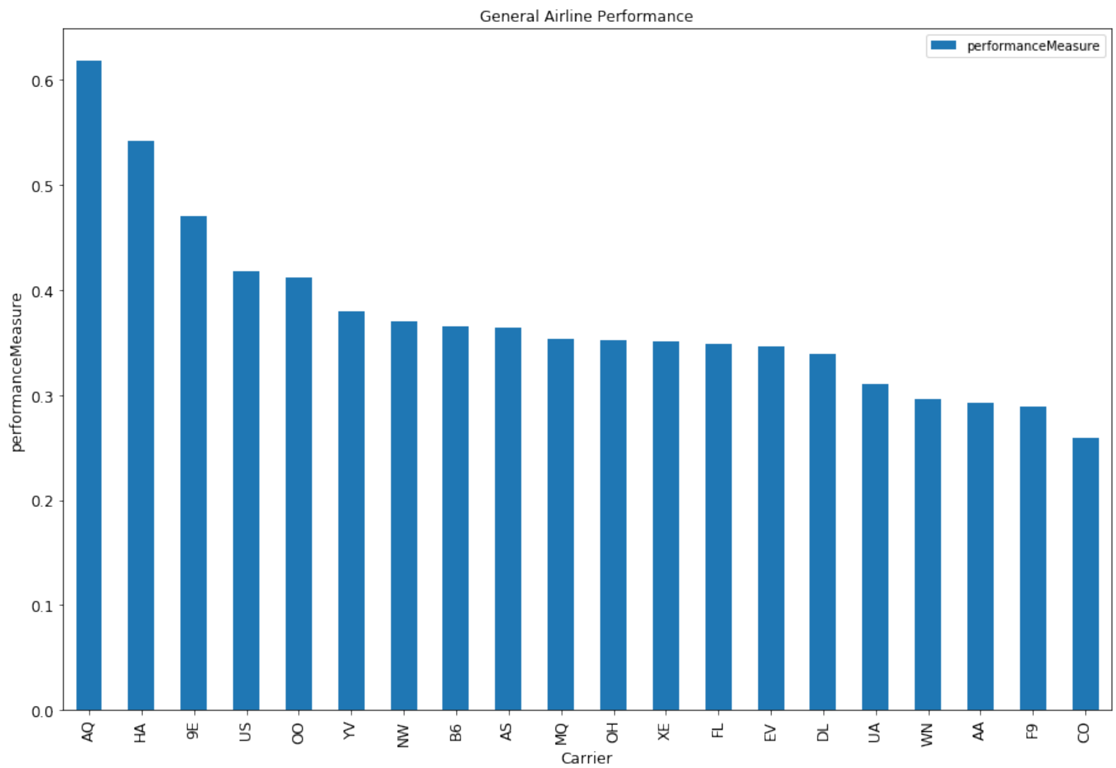
```
                alias("numberOfCancelledFlights"),\
            round(col("isCancelled") / col("numberOfFlights")*100,2).\
                alias("cancellationPct"),\
            round(col("isCompleted") / col("numberOfFlights")*100,2).\
                alias("completedPct")\
          ).sort(desc("performanceMeasure")).toPandas()

# More columns exist, only printing 8
pd.set_option("display.max_columns",8)
carrierPerformanceTable.head(5)
```

Out[75]:

```
  UniqueCarrier  performanceMeasure  depOnTimePct  arrOnTimePct      ...       \
0            AQ            0.618103         82.27         75.54      ...
1            HA            0.542419         78.55         69.70      ...
2            9E            0.470375         73.87         65.45      ...
3            US            0.418076         67.37         62.97      ...
4            OO            0.411363         68.59         61.32      ...

   MaxDepDelay  numberOfCancelledFlights  cancellationPct  completedPct
0          336                        42             0.54         99.46
1          963                       570             0.92         99.08
2         1127                      7100             2.71         97.29
3          886                      6582             1.45         98.55
4          996                     12436             2.19         97.81

[5 rows x 13 columns]
```

In [17]:

```
carriers=carrierPerformanceTable['UniqueCarrier'].tolist()

ax = carrierPerformanceTable[['performanceMeasure']].\
    plot(kind='bar', title ="General Airline Performance",\
         figsize=(15, 10),\
         legend=True,\
         fontsize=12)

ax.set_xticklabels(carriers)
ax.set_xlabel("Carrier", fontsize=12)
ax.set_ylabel("performanceMeasure", fontsize=12)
plt.show()
```



So, it looks like this years price for best airline goes to AQ !

2.1.6 6. Which airport performed the worst seen from a customer perspective?

Lets do the same kind of analysis on airports, eg. which one performes worst, as seen from a customert's viewpoint. Airport performance from a customert's viewpoint could be many things. Some characteristicts could also depend on wether the airport is a origination or a destination for a given flight. Here, wet'll focust on destination airports. We have airport data defined already:

```
In [18]: airports.show(2)
```

```
+----+--------------------+-----------+-----+-------+-----------+------------+
|iata|             airport|       city|state|country|        lat|        long|
+----+--------------------+-----------+-----+-------+-----------+------------+
| 00M|            Thigpen |Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal| Livingston|   TX|    USA|30.68586111|-95.01792778|
+----+--------------------+-----------+-----+-------+-----------+------------+
only showing top 2 rows
```

```
In [19]: # Join flights with airports to get destination airport name
         destinationAirports=airports.select(col("iata"),col("airport")).\
             withColumnRenamed("iata","destIata").withColumnRenamed("airport","destAirport")
```

```
# Join with airports to get origination airport info
flightsWithAirports = flights.join(destinationAirports,\
                                   flights.Dest == destinationAirports.destIata).\
    alias("flightsWithDestinationAirports").\
join(airports,flights.Origin == airports.iata)

# Pretty print the first 10 rows (and 20 columns) using pandas
pd.set_option('display.max_columns', 20)
flightsWithAirports.select("Year",\
                           "Month",\
                           "DayOfMonth",\
                           "UniqueCarrier",\
                           "airport",\
                           "destAirport").\
    limit(10).toPandas()
```

```
Out[19]:    Year  Month  DayOfMonth UniqueCarrier                         airport  \
         0  2008      1           3            WN  Washington Dulles International
         1  2008      1           3            WN  Washington Dulles International
         2  2008      1           3            WN       Indianapolis International
         3  2008      1           3            WN       Indianapolis International
         4  2008      1           3            WN       Indianapolis International
         5  2008      1           3            WN       Indianapolis International
         6  2008      1           3            WN       Indianapolis International
         7  2008      1           3            WN       Indianapolis International
         8  2008      1           3            WN       Indianapolis International
         9  2008      1           3            WN       Indianapolis International

                               destAirport
         0             Tampa International
         1             Tampa International
         2  Baltimore-Washington International
         3  Baltimore-Washington International
         4  Baltimore-Washington International
         5        Jacksonville International
         6            McCarran International
         7            McCarran International
         8         Kansas City International
         9         Kansas City International
```

As mentioned, destination airport performance could be many things. Here, we'll focus on wether flights arrive or not. This is from a presumption, that the reason for flights not arriving is destination airport capacity (need to re-route in difficult situations), its technical equipment to support arriving planes in rough or foggy weather, its location and so on. This is probably not the best estimate of airport performance, but in lack of domain knowledge, this example will do.

```
In [76]: airportPerformanceTable = flightsWithAirports.\
             select(flightsWithAirports.destIata, \
```

```
                    flightsWithAirports.destAirport, \
                    when(flightsWithAirports.Diverted > 0,0).\
                    otherwise(1).alias("hasArrived"),\
                   ).\
             groupBy(flightsWithAirports.destIata, flightsWithAirports.destAirport). \
             agg(sum("hasArrived").alias("hasArrived"),
                 count(lit(1)).alias("numberOfFlights") \
                ). \
             select(col("destIata"), col("destAirport"), \
                    col("numberOfFlights"),\
                    (round(col("hasArrived") / col("numberOfFlights")*100,2))\
                       .alias("completedPct")\
                   ).sort(asc("completedPct")).limit(10).toPandas()
```

```
In [77]: airportPerformanceTable
```

```
Out[77]:   destIata                   destAirport  numberOfFlights  completedPct
        0       OGD                Ogden-Hinckley                2          0.00
        1       CYS                      Cheyenne                2          0.00
        2       OME                          Nome             1090         95.96
        3       TEX            Telluride Regional              194         96.91
        ..      ...                           ...              ...           ...
        6       TWF  Joslin Field - Magic Valley             1788         97.76
        7       SUN             Friedman Memorial             2905         97.80
        8       PSG  James C. Johnson Petersburg               727         98.07
        9       HHH                   Hilton Head              836         98.09

        [10 rows x 4 columns]
```

It looks like there is actually two airports, that have zero-performance by this definition. Two out of two flights have not arrived at the airport. After these, * Nome * Telluride Regional * Ralph Wien Memorial

perform the worst

2.1.7 7. On appserver2 (and possibly your laptop), these files are just stored as ordinary files in the OSmanaged file system. How would they be stored in HDFS running on a cluster? Which advantages/disadvantages would that give?

The HDFS (Haddop Distributed File System) is a distributed filesystem that supports parallellism in file reading/writing on multiple machines in a cluster. This means, that every "logical" file is split into partitions, that are placed on different machines on local storage. This gives us the following benefits: * Reading the whole logical file can be done in parallel by individual machines * Having the partitioned data on local storage, some transformnations can be performed directly on the local partition of data, thus performing them in parallel across machines. * Being able to store files that are larger than any single local harddrive * Fault-tollerance, since all partitions are replicated three (default) times on different nodes

The partitioning scheme and replication however presents a choice between: * Consistency, Availability and Partition tolerance (CAP Theorem)

This means, that if partitioning tolerance is given in HDFS (meaning, that if one partition-replica is corrupted, the system will still be running), we need to chose between consistency and availability. HDFS offers consistency - thus, we can run into availability-issues, since a write to a file means, that to ensure consistency, this write needs to be replicated to other replicas before being able to guarantee a consistent read of the same file. If a network (or other) failure prevents this replication, then the system is down.

Basically, this means, that the HDFS is not a high-availability system, because it gives priority to consistency.

2.1.8 Clustering

The clustering exercise hints the use of weather data. The sequence below downloads, uncompresses and moves relevant files contaning weather information from US-airports in 2008.

```
In [22]: import urllib.request
         import zipfile
         import os

         def downloadAndUnzip(url, filename):
             downloadFile=url+filename
             targetFile="./data/downloadStaging/"+filename
             print("Downloading and upzipping: "+downloadFile)
             urllib.request.urlretrieve(downloadFile, "./data/downloadStaging/"+filename)
             zip_ref = zipfile.ZipFile(targetFile, 'r')
             zip_ref.extractall("./data/downloadStaging")
             zip_ref.close()
             # Cleanup
             os.system('cp ./data/downloadStaging/*daily.txt ./data/weather/')
             os.system('cp ./data/downloadStaging/*station.txt ./data/weather/')
             os.system('rm ./data/downloadStaging/*')


         years=["2008"]
         months=["01","02","03","04","05","06","07","08","09","10","11","12"]
         for year in years:
             for month in months:
                 downloadAndUnzip("https://www.ncdc.noaa.gov/orders/qclcd/","QCLCD"\
                                  +year+month+".zip")
```

```
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200801.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200802.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200803.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200804.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200805.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200806.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200807.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200808.zip
```

```
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200809.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200810.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200811.zip
Downloading and upzipping: https://www.ncdc.noaa.gov/orders/qclcd/QCLCD200812.zip
```

Weatherdata is defined in Spark context and below is the first couple of observations.

```
In [23]: weather = spark.read.csv("./data/weather/*daily.txt",\
                                  header=True,\
                                  inferSchema=True,\
                                  nullValue='NA')
```

```
In [79]: pd.set_option('display.max_columns', 8)
         weatherPD = weather.limit(10).toPandas()
         weatherPD
```

```
Out[79]:     WBAN  YearMonthDay Tmax TmaxFlag     ...     Max2Speed Max2SpeedFlag  \
         0   3013      20080101   33              ...            14
         1   3013      20080102   30              ...            10
         2   3013      20080103   50              ...            14
         3   3013      20080104   56              ...            15
         ..   ...           ...  ...      ...     ...           ...           ...
         6   3013      20080107   45              ...            21
         7   3013      20080108   40              ...            16
         8   3013      20080109   55              ...            33
         9   3013      20080110   43              ...            15

            Max2Dir Max2DirFlag
         0      250
         1      210
         2      260
         3      230
         ..     ...         ...
         6      060
         7      280
         8      350
         9      330

         [10 rows x 50 columns]
```

We need a reference table to tie together weatherdata and airport information, which is present on the "station.txt" files.

```
In [25]: stations = spark.read.option("delimiter", "|").option("header", "True").\
             csv('./data/weather/*station.txt')
```

```
In [26]: ## pd.set_option('display.max_columns', 250)
         sqlContext.registerDataFrameAsTable(stations, "stationsTable")
```

```
callSigns=sqlContext.sql("SELECT distinct WBAN as stationWBAN,"+\
                         "CallSign from stationsTable").\
    persist()

callSigns.limit(2).toPandas()
```

```
Out[26]:   stationWBAN CallSign
         0       03041      MYP
         1       04815      228
```

Wet'll join flight data and weather-station data to translate IATA callsign to WBAN, which is a key in weatherdata. Also, wet'll construnct a "yearMonthDay" column, that will be used for joining later on

```
In [80]: # Start by joining flights with station data to translate IATA-code to WBAN code
         flightsWithStations = flights.join(callSigns,\
                                           flights.Origin==callSigns.CallSign,\
                                           'left_outer').\
             withColumn("yearMonthDay",(concat(col('Year'),\
                                              lpad(col('Month'), 2, '0'),\
                                              lpad(col('DayofMonth'), 2, '0'))).\
             cast("Integer"))

         print("Lets check, if all flights have station information - "+\
               "remember, we have outerjoined")

         print("Looks like we loose flights from the following destinations: ")
         flightsWithStations.where(col('stationWBAN').\
                                  isNull()).\
         groupby(col("Origin")).count().show()

         flightsWithStations.limit(10).toPandas()
```

```
Lets check, if all flights have station information - remember, we have outerjoined
Looks like we loose flights from the following destinations:
+------+-----+
|Origin|count|
+------+-----+
|   PSE|  755|
|   SCE|  645|
|   HHH|  836|
|   FCA| 2762|
|   CLD| 2303|
+------+-----+
```

```
Out[80]:     Year  Month  DayofMonth  DayOfWeek       ...        LateAircraftDelay  \
         0   2008      1           3          4       ...                      NaN
```

```
1    2008       1           3          4       ...                NaN
2    2008       1           3          4       ...                NaN
3    2008       1           3          4       ...                NaN
..    ...     ...         ...        ...       ...                ...
6    2008       1           3          4       ...               47.0
7    2008       1           3          4       ...                NaN
8    2008       1           3          4       ...                NaN
9    2008       1           3          4       ...                NaN

    stationWBAN  CallSign  yearMonthDay
0         93738       IAD      20080103
1         93738       IAD      20080103
2         93819       IND      20080103
3         93819       IND      20080103
..          ...       ...           ...
6         93819       IND      20080103
7         93819       IND      20080103
8         93819       IND      20080103
9         93819       IND      20080103

[10 rows x 32 columns]
```

Lets join the weather…

```
In [28]: # Now, lets join the weather information for the originating airport:
         # This might be a tough one, joining 7 mill flights with 360K rows of weatherdata
         # How about the explain plan ?
         flightsWithOriginWeather=flightsWithStations.\
             join(weather,(flightsWithStations.stationWBAN==weather.WBAN) & \
                          (flightsWithStations.yearMonthDay == weather.YearMonthDay),\
                  'left_outer').\
             drop('YearMonthDay')

         print("This join looses the following number of rows:")
         flightsWithOriginWeather.where(col('WBAN').isNull()).count()
```

```
This join looses the following number of rows:
```

```
Out[28]: 71613
```

From the above query, it looks like we loose

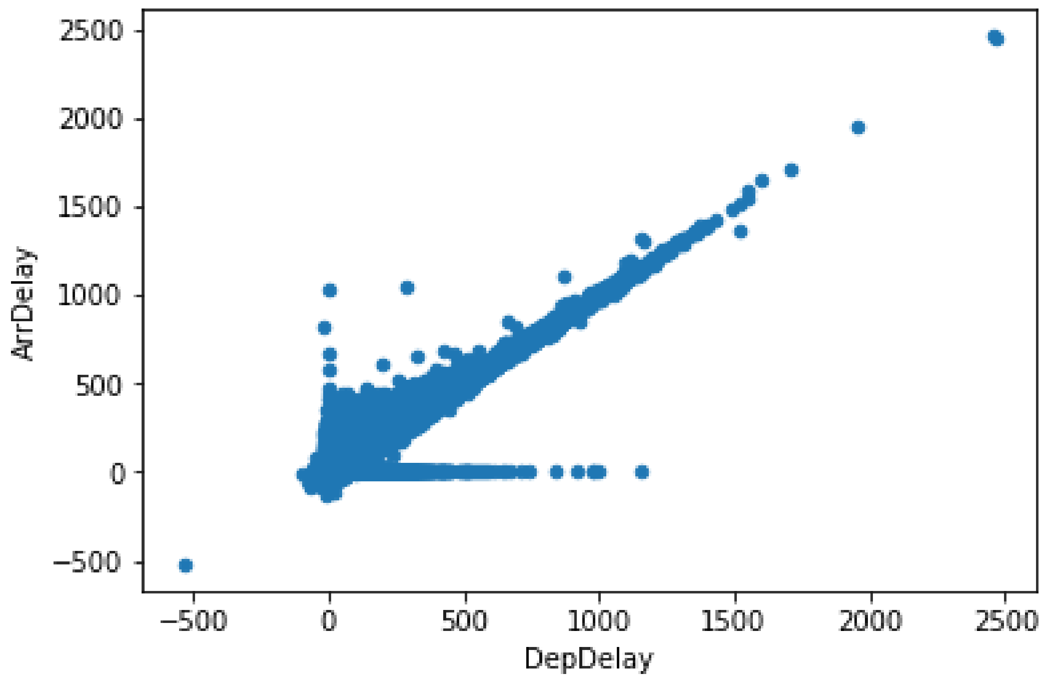
PySPark ML library - in contrast to Scikit-learn - requires features to be assembled in one column. The ML library supplies a method for doing so, vectorAssembler, which wet'll see below. Here, no NULL column are allowed, so wet'll replace them with 0 (as in "no delay").

```
In [29]: dfForClustering = flightsWithOriginWeather.where(col('WBAN').isNotNull()).\
             select(col('UniqueCarrier'),col('DepDelay'),col('ArrDelay')).na.fill(0)
```

Since we'll be clustering with a focus on delay, lets take a look at the two (main) types of delay, departure- and arrivaldelay.

```
In [30]: dfForClustering.select(col('DepDelay'),\
                                col('ArrDelay'))\
         .toPandas().plot.scatter(x='DepDelay',\
                                  y='ArrDelay')
```

```
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddffd64630>
```



Some observations here: * It looks like departure- and arrivaledelay is very much correlated (which makes sense to some degree) * It looks like we have observations where departuredelay is 1.000 minutes, but arrival-delay is 0 (and the otherway around. That sounds unlikely, however this could be poor dataquality, where either departuredelay or arrivaldelay are not present. Remember, that in these cases, we chose to set them to 0. We could have chosen to omit these flights instead, but that would remove delayed flights from the data and we'd have to set a threshold on wether to judge a delay as 0 or missing because of poor dataquality. Instead we keep them, and now have no flights without delay-information.

```
In [31]: dfForClustering.where(col('DepDelay').isNull()).count(),\
         dfForClustering.where(col('ArrDelay').isNull()).count()
```

```
Out[31]: (0, 0)
```

A measure of the correlation between the delay-types:

In [32]:
```
from pyspark.ml.stat import Correlation
dfForClustering.stat.corr("DepDelay", "ArrDelay")
```

Out[32]: 0.9265700598211729

We could cluster from a combination of departure- and arrival delay, but that would yield a measure of delay, that is not quite intuitive. As it looks like departure and arrival delays are very much correlated, it would suffice to accept arrivaldelay only as a general measure of delay. This is probably also the most important delay-type, seen from a customer view-point.

Rather that setting up fx low/medium/high delay-groups, we look into the flights data for hidden groups, using the Kmeans clustering method to divide the flights into "natural" delay-groups. As mentioned above, wet'll "assemple" a featurevector (consisting of only one-value, arrivaldelay) for the clustering and call it "features":

In [33]:
```
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
%matplotlib inline
# VectorAssembler does not accept NULL values
features = ['ArrDelay']
assembler = VectorAssembler(inputCols=features, outputCol="features")
baseClusteringDF = assembler.transform(dfForClustering).cache()
baseClusteringDF.limit(10).toPandas()
```

Out[33]:

	UniqueCarrier	DepDelay	ArrDelay	features
0	WN	8	-14	[-14.0]
1	WN	19	2	[2.0]
2	WN	8	14	[14.0]
3	WN	-4	-6	[-6.0]
4	WN	34	34	[34.0]
5	WN	25	11	[11.0]
6	WN	67	57	[57.0]
7	WN	-1	-18	[-18.0]
8	WN	2	2	[2.0]
9	WN	0	-16	[-16.0]

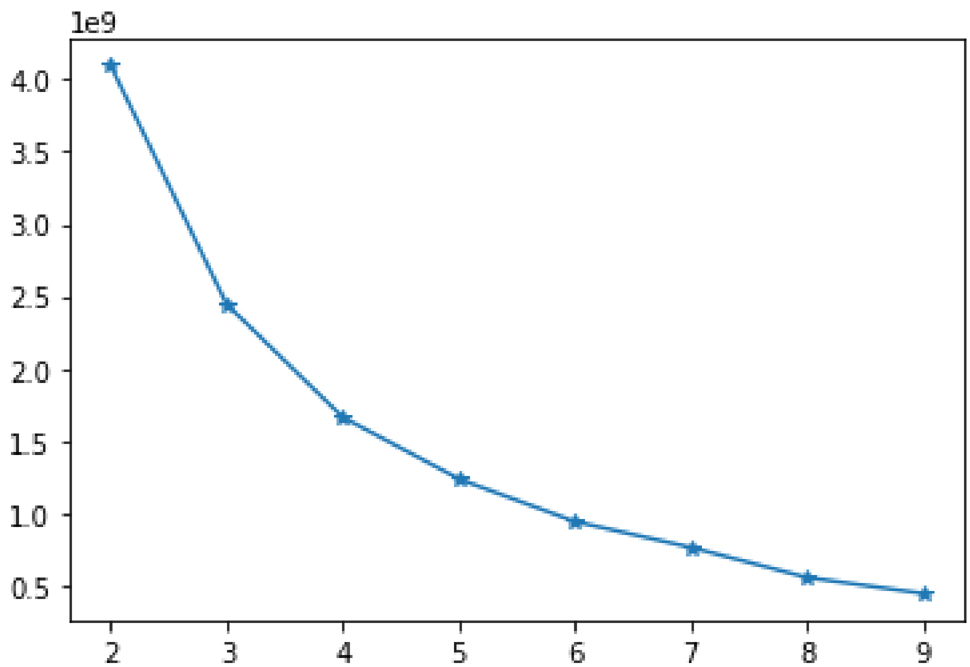
The KMeans algorithm needt to know how many clusters to create in total, which is har to know in advance. Therefor, we can apply the "elbow-method" to see, if theret's any "natural number of clusters". The tqdm library allows ud to create a statusbar when looping, just to get a sence of progree while running:

In [34]:
```
from tqdm import tqdm
elbowDict={}
numberOfClusters = range(2,10)
for cluster in tqdm(numberOfClusters):
    # The following stuff is going on at Spark level
    kmeans = KMeans(k=cluster, seed=1)
    model = kmeans.fit(baseClusteringDF.select('features'))
    WSSSE = model.computeCost(baseClusteringDF.select('features'))
```

```
    # Now, back to Python and append the cost to the dictionary
    elbowDict[cluster]=WSSSE
```

```
100%|| 8/8 [01:17<00:00,  9.66s/it]
```

```
In [35]: import matplotlib.pylab as plt
         lists = sorted(elbowDict.items()) # sorted by key, return a list of tuples
         x, y = zip(*lists) # unpack a list of pairs into two tuples
         plt.plot(x, y, marker='*')
         plt.show()
```



There is no clear "elbow" point, so wet'll choose 5 clusters for the number of delay-groups, and train the model.

```
In [36]: cluster=5
         kmeans = KMeans(k=cluster, seed=1)
         model = kmeans.fit(baseClusteringDF.select('features'))
         # "Predict" the cluster for each flight
         transformed = model.transform(baseClusteringDF)
```

Lets take a look at the groups. The window functions allows us to rank the clusters according to average arival delay, such tha group 1 is leat delayed and group 5 is the most delayed flights:

```
In [37]: from pyspark.sql import functions as F
         from pyspark.sql.window import Window
```

```
delayGroups = transformed.groupBy("prediction").\
agg(avg('ArrDelay').alias('avgArrDelay'), \
    min('ArrDelay').alias('minArrDelay'), \
    max('ArrDelay').alias('maxArrDelay'), \
    sum(lit(1)).alias('numberOfFlights')).\
withColumn('delayGroup',\
           F.row_number().\
           over(Window.partitionBy(lit(1)).\
                orderBy(col("avgArrDelay")))).cache()

delayGroups.toPandas().sort_values(by=['delayGroup'])
```

Out[37]:

```
   prediction  avgArrDelay  minArrDelay  maxArrDelay  numberOfFlights  \
0           0   -11.843752         -519           -3          3279086
1           2     7.662529           -2           29          2739910
2           4    51.709830           30           91           669026
3           1   132.032380           92          213           215473
4           3   297.201704          214         2461            34620

   delayGroup
0           1
1           2
2           3
3           4
4           5
```

We'll join the delaygroups with the flightsdata for later use. Since we are running things in the cloud, we pay by the hour of using the machine, so it would be cost-effective be able to shut down the machine when not using it. Therefor we'll save the intermediate results to disk, to be able to continue the analysis after having restartet.

In [81]:

```
flightsWithDelayGroup = flightsWithOriginWeather.where(col('WBAN').isNotNull()).\
  join(delayGroups,(coalesce(flightsWithOriginWeather.ArrDelay,lit(0)) >=\
                      delayGroups.minArrDelay) & \
                  (coalesce(flightsWithOriginWeather.ArrDelay,lit(0)) <=\
                      delayGroups.maxArrDelay))\
            .cache()
```

In [39]:

```
# Persist to disk to be able to restart
flightsWithDelayGroup.write.mode('overwrite').\
parquet("./data/flightsWithDelayGroup.parquet")

weather.write.mode('overwrite').\
parquet("./data/weather.parquet")

delayGroups.write.mode('overwrite').\
parquet("./data/delayGroups.parquet")
```

In [40]:

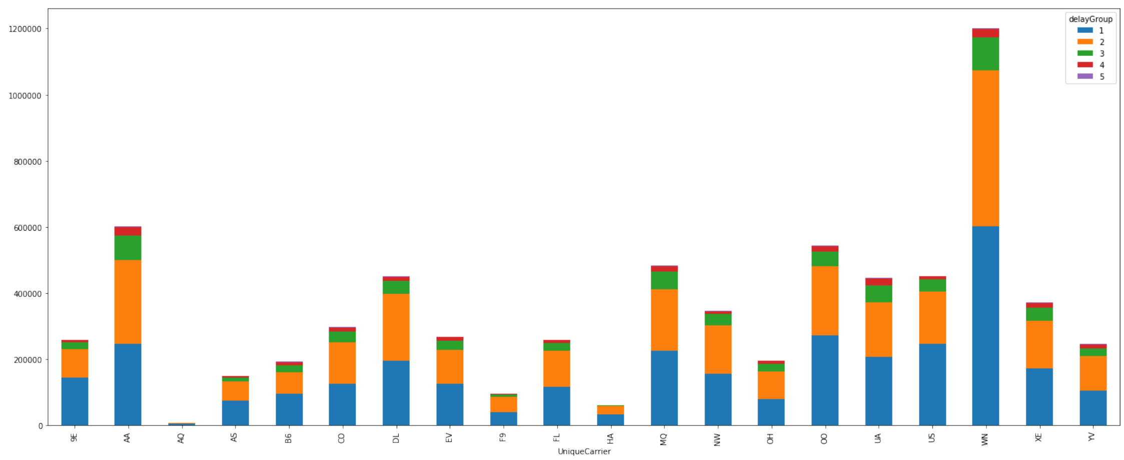
```
# Read after restart
flightsWithDelayGroup=sqlContext.read.parquet("./data/flightsWithDelayGroup.parquet")
```

```
weather=sqlContext.read.parquet("./data/weather.parquet")
delayGroups=sqlContext.read.parquet("./data/delayGroups.parquet")
```

Now, with this information, we can rank the airlines on the basis of arrival-delay-group aswell. Lets stack the flights with delayGroup for each carrier. First wet'll aggregate the carrierinfo in Spark and then use Pandas to plot.

```
In [41]: groupedUniqueCarriers = flightsWithDelayGroup.\
         groupBy(col('UniqueCarrier'),col('delayGroup')).\
         agg(sum(lit(1)).alias('numberOfFlights')).\
         toPandas()

         pt = groupedUniqueCarriers.pivot('UniqueCarrier', 'delayGroup', 'numberOfFlights')
         pt.plot(kind='bar', stacked=True, figsize=(25,10))
```

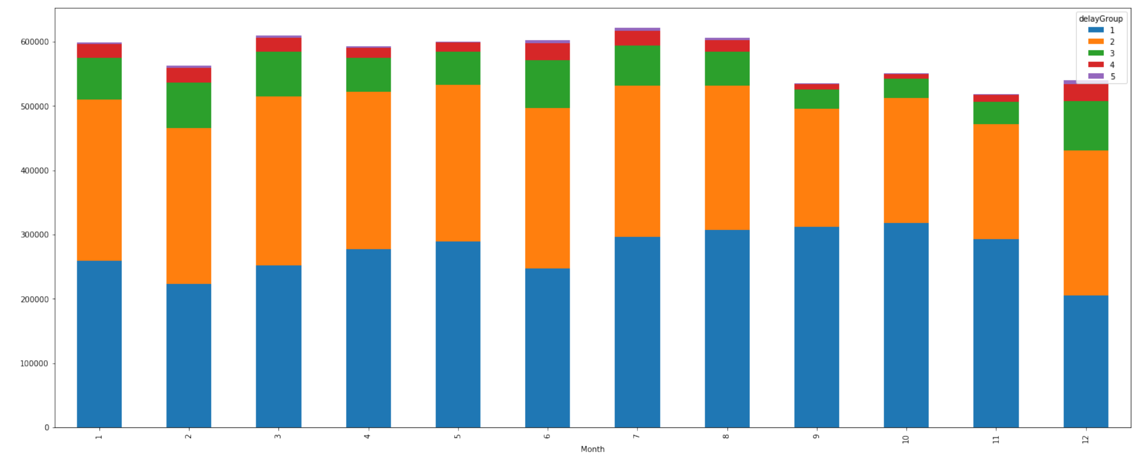
```
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddc0ab5860>
```



Another way would be to see, if any month looks worse than the others:

```
In [42]: groupedUniqueCarriers = flightsWithDelayGroup.\
         groupBy(col('Month'),col('delayGroup')).\
         agg(sum(lit(1)).alias('numberOfFlights')).\
         toPandas()

         pt = groupedUniqueCarriers.pivot('Month', 'delayGroup', 'numberOfFlights')
         pt.plot(kind='bar', stacked=True, figsize=(25,10))
```

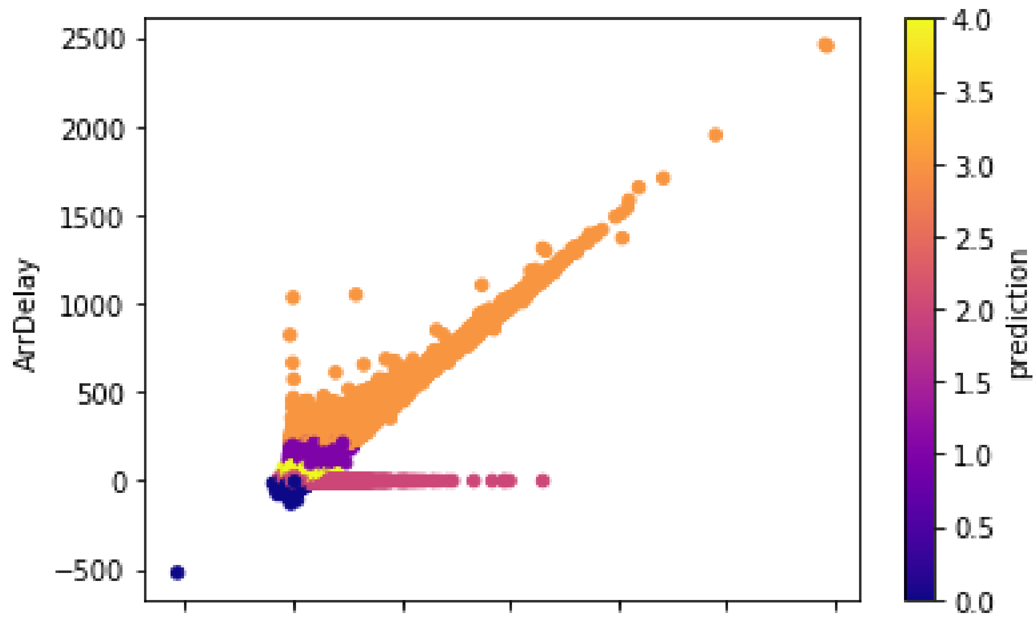
```
Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddffcf7048>
```



Visually, it looks like june and december generally have more arrival delays.

Now, having assigned arrival-delay-group to the flights, we could reprint the flightdelay correlation plot again, now adding the delay-group as color coding.

```
In [43]: transformed = model.transform(baseClusteringDF)
         featuresAndPrediction = transformed.select("DepDelay",\
                                                    "ArrDelay",\
                                                    "prediction")

         featuresAndPredictionPD=featuresAndPrediction.select(col('DepDelay'),\
                                                              col('ArrDelay'),\
                                                              col('prediction')).\
             toPandas()

         featuresAndPredictionPD.plot.scatter(x='DepDelay',\
                                              y='ArrDelay',\
                                              c='prediction',\
                                              cmap='plasma')

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddc0aa9cf8>
```



2.1.9 Classification

In the below section, wet'll build two different classification models to predict arrival-delay and cancellation of any given flight from its characteristics. Wet'll utillize the weather information, but for weather-features to hold any predictive power, atleast they have to have multiple values. If any given column has only one distinct value (or null), it will not be able to hold predictive power, and wet'll omit it from the feature vector. The below count will show us which variables that have only one distinct value:

```
In [44]: #dist=flightsWithDelayGroup.agg(
         dist=weather.agg(
         countDistinct("Tmax"),
         countDistinct("TmaxFlag"),
         countDistinct("Tmin"),
         countDistinct("TminFlag"),
         countDistinct("Tavg"),
         countDistinct("TavgFlag"),
         countDistinct("Depart"),
         countDistinct("DepartFlag"),
         countDistinct("DewPoint"),
         countDistinct("DewPointFlag"),
         countDistinct("WetBulb"),
         countDistinct("WetBulbFlag"),
         countDistinct("Heat"),
         countDistinct("HeatFlag"),
         countDistinct("Cool"),
```

```
        countDistinct("CoolFlag"),
        countDistinct("Sunrise"),
        countDistinct("SunriseFlag"),
        countDistinct("Sunset"),
        countDistinct("SunsetFlag"),
        countDistinct("CodeSum"),
        countDistinct("CodeSumFlag"),
        countDistinct("Depth"),
        countDistinct("DepthFlag"),
        countDistinct("Water1"),
        countDistinct("Water1Flag"),
        countDistinct("SnowFall"),
        countDistinct("SnowFallFlag"),
        countDistinct("PrecipTotal"),
        countDistinct("PrecipTotalFlag"),
        countDistinct("StnPressure"),
        countDistinct("StnPressureFlag"),
        countDistinct("SeaLevel"),
        countDistinct("SeaLevelFlag"),
        countDistinct("ResultSpeed"),
        countDistinct("ResultSpeedFlag"),
        countDistinct("ResultDir"),
        countDistinct("ResultDirFlag"),
        countDistinct("AvgSpeed"),
        countDistinct("AvgSpeedFlag"),
        countDistinct("Max5Speed"),
        countDistinct("Max5SpeedFlag"),
        countDistinct("Max5Dir"),
        countDistinct("Max5DirFlag"),
        countDistinct("Max2Speed"),
        countDistinct("Max2SpeedFlag"),
        countDistinct("Max2Dir"),
        countDistinct("Max2DirFlag")).toPandas()
```

In [83]:
```
pd.set_option("display.max_columns",4)
dist
```

Out[83]:
```
   count(DISTINCT Tmax)  count(DISTINCT TmaxFlag)  \
0                   169                         2

              ...              count(DISTINCT Max2Dir)  \
0             ...                                   43

   count(DISTINCT Max2DirFlag)
0                            1

[1 rows x 48 columns]
```

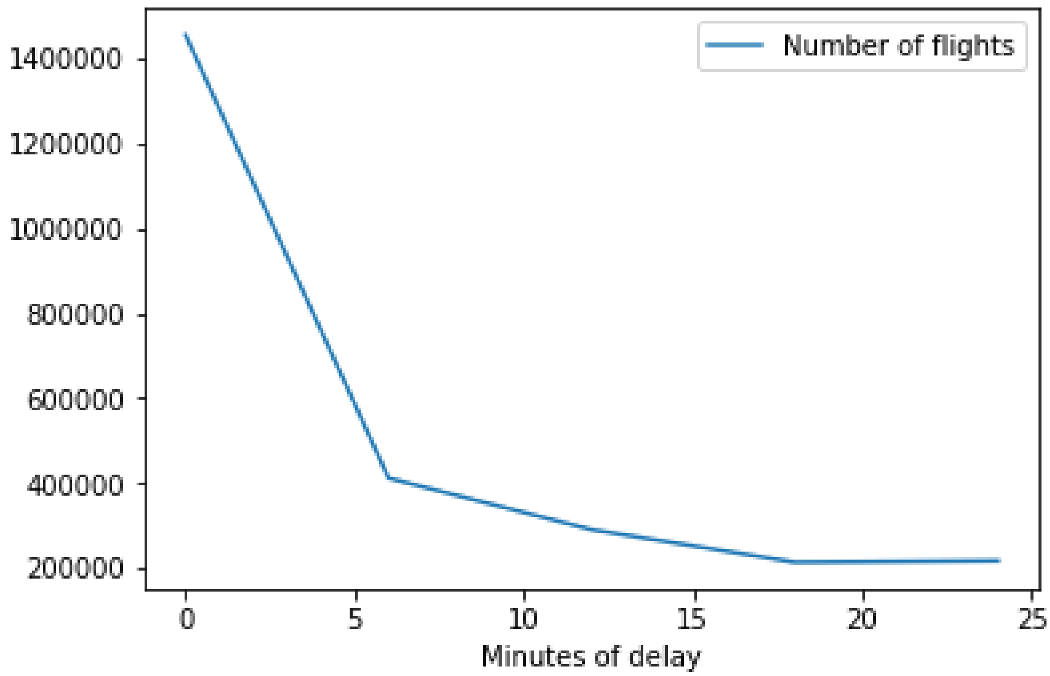
Fx, we should omit the features Max2DirFlag etc... Now, remember the delay-groups ?

```
In [46]: delayGroups.toPandas().sort_values(by=['delayGroup'])
```

```
Out[46]:    prediction  avgArrDelay  minArrDelay  maxArrDelay  numberOfFlights  \
         0           0   -11.843752         -519           -3          3279086
         1           2     7.662529           -2           29          2739910
         2           4    51.709830           30           91           669026
         3           1   132.032380           92          213           215473
         4           3   297.201704          214         2461            34620

            delayGroup
         0           1
         1           2
         2           3
         3           4
         4           5
```

Since the first delaygroup holds flights that are not really delayed and considering, that 7 minutes (the avg of group 2) might not be that big a deal, wet'll invent a new concept of arrivaldelay, eg. delayedStatus, which applies to flights that are in the first two lower groups - eg delayed atmost 29 minutes and being in a group with average delay of 7,6 minutes or below. As can be seen from the below plot, most of the fligts in delayGroup 2 are below 10 minutes anyway.

```
In [47]: #flightsWithDelayGroup.limit(3).toPandas()
         histogramData = flightsWithDelayGroup.where(flightsWithDelayGroup.delayGroup==2).\
         withColumn("rnd", round(flightsWithDelayGroup["arrDelay"]/5)*5)

         histogram = histogramData.select('rnd').rdd.flatMap(lambda x: x).histogram(5)

         # Loading the Computed Histogram into a Pandas Dataframe for plotting
         pd.DataFrame(
             list(zip(*histogram)),
             columns=['Minutes of delay', 'Number of flights']
         ).set_index(
             'Minutes of delay'
         ).plot(kind='line');
```



So, wet'll create a new binary classification label column, that holds 0 (not delayed) if the delaygroup is 1 or 2 and 1 (delayed) for other flights

```
In [48]: # Wet'll create a binary classification target (delayedStatus)
         # Wet'll consider delayGroup 1 and 2 as no delay, since it is such a small delay
         # (avg 7,66 mins and <29 mins)
         flightsWithDelayStatus=flightsWithDelayGroup.\
             withColumn('delayedStatus',when(flightsWithDelayGroup.delayGroup > 2, 1).\
                                                   otherwise(0))

         flightsWithDelayStatus.persist()

Out[48]: DataFrame[Year: int, Month: int, DayofMonth: int, DayOfWeek: int, DepTime: int, CRSDe

In [49]: flightsWithDelayStatus.groupBy(col('delayedStatus')).count().show()

+-------------+-------+
|delayedStatus|  count|
+-------------+-------+
|            1| 919119|
|            0|6018996|
+-------------+-------+
```

As mentioned, the Spark machine learning library (ML for dataframes, Mllib for RDDt's) expects features to be assembled in a feature vector. As seen earlier, this can be done using the vectorAssembler. However theret's also another tool for this operation: From https://spark.apache.org/docs/1.6.2/ml-features.html#rformula : RFormula selects columns specified by an R model formula. It produces a vector column of features and a double column of labels. Like when formulas are used in R for linear regression, string input columns will be one-hot encoded, and numeric columns will be cast to doubles. If not already present in the DataFrame, the output label column will be created from the specified response variable in the formula.

We create a feature vector column for the classifier along with the label. Once this is done, wet'll drop all other columns, since we do not want to carry all this data around for no use.

```
In [50]: # https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
         from pyspark.ml.feature import RFormula
         formula = RFormula(
             formula="delayedStatus ~ "+\
             "Tmax + TmaxFlag + Tmin + TminFlag + Tavg + Depart + DewPoint + "+\
             "WetBulb + Heat + Cool + Sunrise + Sunset + CodeSum  + Depth + SnowFall + "+\
             "SnowFallFlag + PrecipTotal + PrecipTotalFlag + StnPressure + SeaLevel + "+\
             "ResultSpeed + ResultDir + AvgSpeed + Max5Speed + Max5SpeedFlag + Max5Dir +"+\
             "Max2Speed + Max2SpeedFlag + Max2Dir",
             # Lets try non weather data
             #formula="delayedStatus ~ DepDelay + DepTime + Distance ",
             featuresCol="features",
             labelCol="label")

         output = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('NA')).\
                   transform(flightsWithDelayStatus.na.fill(0).na.fill('NA')).\
                   select("features","label")
```

Splitting into training- and test data. Lets also count the occurances of labels in the training data - this gives us an idea of how balanced the dataset is. We might want to balance it before training, so not to induce artificial bias towards the majority class.

```
In [51]: (trainingData, testData) = output.randomSplit([0.8,0.2], seed = 13234 )
         trainingData.groupBy(col('Label')).count().show()
```

```
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|4816809|
|  1.0| 736065|
+-----+-------+
```

Even after having redefined the delay concept (into delayStatus), we have an unbalanced trainingset, so wet'll downsample the majority class to get a balanced set. This way, we will avoid the bias in training the model. The testset however should resemble unseen data, thus wet'll keep that unbalanced. Also, wet'll reduce the data to 1/10th to support faster processing time.

```
In [52]: # Downsampling on-time flights (traininset only) to get a balanced dataset
         from pyspark.sql import DataFrame
```

```
trainingDataBalanced = trainingData.where(col('label')==0).\
sample(False,\
       (735037/4815457)/10,
       42).\
    unionAll(trainingData.where(col('label')==1).\
            sample(False,\
                   1/10,\
                   42))

trainingDataBalanced.groupBy(col('Label')).count().show()
```

```
+-----+-----+
|Label|count|
+-----+-----+
|  0.0|74017|
|  1.0|73419|
+-----+-----+
```

Saving the training set, so that we can restart the process from here later on.

```
In [53]: # Important to persist before training model, because of the iterative nature of
         trainingDataBalanced.persist()

Out[53]: DataFrame[features: vector, label: double]

In [54]: #flightsWithDelayGroup.limit(10).toPandas()
         # Persist to disk to be able to restart
         trainingDataBalanced.write.mode('overwrite').\
         parquet("./data/trainingDataBalanced.parquet")

         testData.write.mode('overwrite').\
         parquet("./data/testData.parquet")

In [55]: #flightsWithDelayGroup.limit(10).toPandas()
         # Read disk-persistent datasets to restart
         trainingDataBalanced=sqlContext.read.\
         parquet("./data/trainingDataBalanced.parquet").persist()

         testData=sqlContext.read.\
         parquet("./data/testData.parquet").persist()

In [56]: #trainingDataBalanced.columns
         # Important, when training model, because of the iterative nature
         # of gridsearching and training
         trainingDataBalanced.persist()

Out[56]: DataFrame[features: vector, label: double]
```

```
In [57]: # Cross validation - too expensive, takes too long, but does work
         #from pyspark.ml.evaluation import BinaryClassificationEvaluator
         #from pyspark.ml.classification import DecisionTreeClassifier
         #from pyspark.ml import Pipeline
         # Evaluate model
         #evaluator = BinaryClassificationEvaluator()
         # Create ParamGrid for Cross Validation
         #from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

         #tree = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDepth=5,
         #                              minInstancesPerNode=20, impurity="gini")


         #paramGrid = (ParamGridBuilder()
         #              .addGrid(tree.maxDepth, [1, 2, 6, 10])
         #              .addGrid(tree.minInstancesPerNode, [10, 20, 40])
         #              .build())

         # Create 5-fold CrossValidator
         #cv = CrossValidator(estimator=tree,\
         #                     estimatorParamMaps=paramGrid,\
         #                     evaluator=evaluator,\
         #                     numFolds=5)

         # Run cross validations
         #cvModel = cv.fit(trainingDataBalanced.select("features","label"))

         #print("numNodes = ", cvModel.bestModel.numNodes)
         #print("depth = ", cvModel.bestModel.depth)
         #print("depth = ", cvModel.bestModel.minInstancesPerNode)
```

We will be building a decision tree for classifying flights and preficting delays. The decision tree algorithm takes multiple hyper-parameters, and to tune these, wet'll do a train/validation split. Wet'll use and existing method for this operation.

```
In [58]: from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
         from pyspark.ml.evaluation import BinaryClassificationEvaluator
         from pyspark.ml.classification import DecisionTreeClassifier
         evaluator = BinaryClassificationEvaluator()

         tree = DecisionTreeClassifier(labelCol="label",\
                                       featuresCol="features",\
                                       impurity="gini")

         paramGrid = (ParamGridBuilder()
                      .addGrid(tree.maxDepth, [1, 2, 6, 10])
                      .addGrid(tree.minInstancesPerNode, [10, 20, 40])
                      .build())
```

```python
# Create trainValidationSplit
tvs = TrainValidationSplit(estimator=tree,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           # 80% of the data will be used for training,
                           # 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(trainingDataBalanced.select("features","label"))
```

In [59]:
```python
treeModel = model.bestModel
treeModel
```

Out[59]: DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4080800c7c2646f2c801) of d

Having found the best model, wet'll test it by classifying testdata and finding different performance measures, eg. aROC, precision, recall and printing the confusion matrix.

In [61]:
```python
# The not so pretty Spark-way of printing the confusion matrix
import matplotlib.pyplot as plt
import numpy as np

from pyspark.mllib.evaluation import MulticlassMetrics, BinaryClassificationMetrics
from pyspark.mllib.util import MLUtils
# Get predictions

predictions = treeModel.transform(testData).select('prediction','label')

# Turn the array into an RDD to
predRDD = predictions.rdd.map(lambda p: (float(p.prediction), p.label)).cache()

# Wet'll define two metrics object, to extract the confusion matrix from one
# and the ROCAUC from the other

metrics = BinaryClassificationMetrics(predRDD)
mcMetrics = MulticlassMetrics(predictions.rdd)
mcMetrics.confusionMatrix().toArray().transpose()
```

Out[61]:
```
array([[741400.,  79989.],
       [460787., 103065.]])
```

In [62]:
```python
# Summary stats
print("ROC = %s" % metrics.areaUnderROC)
```

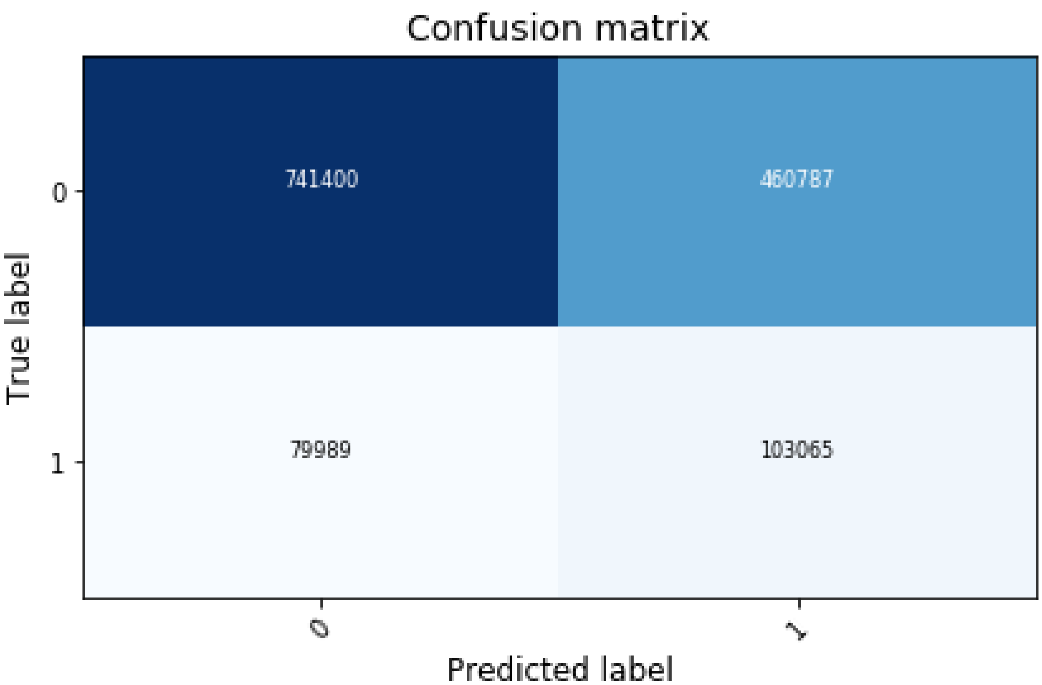
```
ROC = 0.5898699808136934
```

As seen, Spark can compute the performance measures for the model, but for pretty printing, wet'll go to Pandas and matplotlib, as before. Below code is borrowed from : https://stackoverflow.com/questions/44054534/confusion-matrix-error-when-array-dimensions-are-of-size-3

In [63]:
```
import matplotlib.pyplot as plt
import itertools
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import numpy as np

def pretty_print_conf_matrix(y_true, y_pred,
                             classes,
                             normalize=False,
                             title='Confusion matrix',
                             cmap=plt.cm.Blues):
    """
    Mostly stolen from:
    http://scikit-learn.org/stable/auto_examples/model_selection/
    plot_confusion_matrix.html#
    sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py

    Normalization changed, classification_report stats added below plot
    """

    cm = confusion_matrix(y_true, y_pred)
    #cm = confArray

    # Configure Confusion Matrix Plot Aesthetics (no text yet)
    plt.imshow(cm, interpolation='nearest', cmap=cmap, aspect='auto')
    plt.title(title, fontsize=14)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    plt.ylabel('True label', fontsize=12)
    plt.xlabel('Predicted label', fontsize=12)

    # Calculate normalized values (so all cells sum to 1) if desired
    if normalize:
        cm = np.round(cm.astype('float') / cm.sum(),2) #(axis=1)[:, np.newaxis]

    # Place Numbers as Text on Confusion Matrix Plot
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 verticalalignment = 'bottom',
                 color="white" if cm[i, j] > thresh else "black",
```

```
                    fontsize=8)


    # Add Precision, Recall, F-1 Score as Captions Below Plot
    rpt = classification_report(y_true, y_pred)
    rpt = rpt.replace('avg / total', '      avg')
    rpt = rpt.replace('support', 'N Obs')

    plt.annotate(rpt,
                 xy = (0,0),
                 xytext = (-50, -200),
                 #xytext = (0, 0),
                 xycoords='axes fraction', textcoords='offset points',
                 fontsize=12, ha='left')

    # Plot
    plt.tight_layout()
```

In [64]:
```
# Pretty printing with Pandas and MatPlotLib
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.mllib.util import MLUtils
predictions = treeModel.transform(testData).select('prediction','label')

true=predictions.select('label').toPandas() #Serializing to native Python (Pandas)
                                             # dataframe
predicted=predictions.select('prediction').toPandas()
pretty_print_conf_matrix(true,\
                         predicted,\
                         classes=[0,1],\
                         normalize=False,\
                         title='Confusion matrix',\
                         cmap=plt.cm.Blues)
```



	precision	recall	f1-score	N Obs
0.0	0.90	0.62	0.73	1202187
1.0	0.18	0.56	0.28	183054
avg	0.81	0.61	0.67	1385241

```
In [65]: print("ROC = %s" % metrics.areaUnderROC)
```

```
ROC = 0.5898699808136934
```

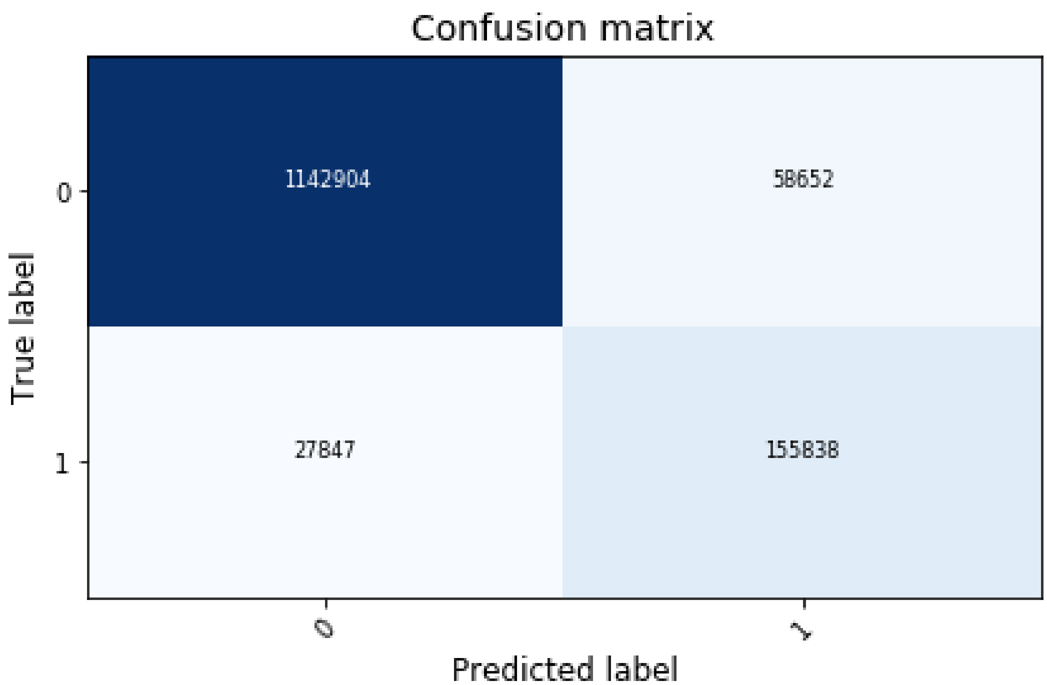
Well, it doesn't look like we can predict the arrival delays from weather-data alone very well. As seen earlier, departure delay and arrival delay is very much correlated, which must also mean, that departure delay would be a good predictor - lets try that, just for the fun of it. So, repeating all the steps from above, with a different RFormula feature vector genertor, now including DepDelay and some other non-weather features.

```
In [66]: import datetime
         print(str(datetime.datetime.now())+": Generating feature-vector")
```

```
formula = RFormula(
    # Lets try non weather data
    formula="delayedStatus ~ DepDelay + DepTime + Distance + DayOfWeek",
    featuresCol="features",
    labelCol="label")

output = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('NA')).\
         transform(flightsWithDelayStatus.na.fill(0).na.fill('NA')).\
         select("features","label")

print(str(datetime.datetime.now())+": Splitting train/testdata")
(trainingData, testData) = output.randomSplit([0.8,0.2], seed = 13234 )
trainingData.groupBy(col('Label')).count().show()

print(str(datetime.datetime.now())+": Downsampling majority class")

# Downsampling on-time flights (traininset only) to get a balanced dataset
trainingDataBalanced = trainingData.where(col('label')==0).\
                                sample(False, (735037/4815457)/10, 42).\
                       unionAll(trainingData.where(col('label')==1).\
                                sample(False, 1/10, 42))
trainingDataBalanced.groupBy(col('Label')).count().show()

print(str(datetime.datetime.now())+": Searching for best model")

tree = DecisionTreeClassifier(labelCol="label",\
                              featuresCol="features",\
                              impurity="gini")

paramGrid = (ParamGridBuilder()
             .addGrid(tree.maxDepth, [1, 2, 6, 10])
             .addGrid(tree.minInstancesPerNode, [10, 20, 40])
             .build())

# Create trainValidationSplit
tvs = TrainValidationSplit(estimator=tree,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           # 80% of the data will be used for training,
                           # 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(trainingDataBalanced.select("features","label"))
treeModel = model.bestModel

print(str(datetime.datetime.now())+": Predicting... and computing metrics")
predictions = treeModel.transform(testData).select('prediction','label')
```

```python
# Turn the array into an RDD to
predRDD = predictions.rdd.map(lambda p: (float(p.prediction), p.label)).cache()

# We'll define two metrics object, to extract the confusion matrix from one
# and the ROCAUC from the other
metrics = BinaryClassificationMetrics(predRDD)
mcMetrics = MulticlassMetrics(predictions.rdd)
mcMetrics.confusionMatrix().toArray().transpose()

# Summary stats
print(str(datetime.datetime.now())+": ROC = %s" % metrics.areaUnderROC)

print(str(datetime.datetime.now())+": Building pretty confusion matrix")
predictions = treeModel.transform(testData).select('prediction','label')
true=predictions.select('label').toPandas() #Serializing to native Python (Pandas) da
predicted=predictions.select('prediction').toPandas()

pretty_print_conf_matrix(true,\
                         predicted,\
                         classes=[0,1],\
                         normalize=False,\
                         title='Confusion matrix',\
                         cmap=plt.cm.Blues)
```

```
2018-06-10 17:45:02.295726: Generating feature-vector
2018-06-10 17:45:02.511755: Splitting train/testdata
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|4817440|
|  1.0| 735434|
+-----+-------+

2018-06-10 17:45:09.024194: Downsampling majority class
+-----+-----+
|Label|count|
+-----+-----+
|  0.0|74029|
|  1.0|73359|
+-----+-----+

2018-06-10 17:45:20.233952: Searching for best model
2018-06-10 17:46:39.665679: Predicting... and computing metrics
2018-06-10 17:46:55.908789: ROC = 0.8997923503966694
2018-06-10 17:47:04.324146: Building pretty confusion matrix
```




```
         precision    recall  f1-score   N Obs

   0.0      0.98      0.95      0.96   1201556
   1.0      0.73      0.85      0.78    183685

   avg      0.94      0.94      0.94   1385241
```

That was a lot better - but as mentioned, we did expect that, since departure delay was included in the feature-vector.

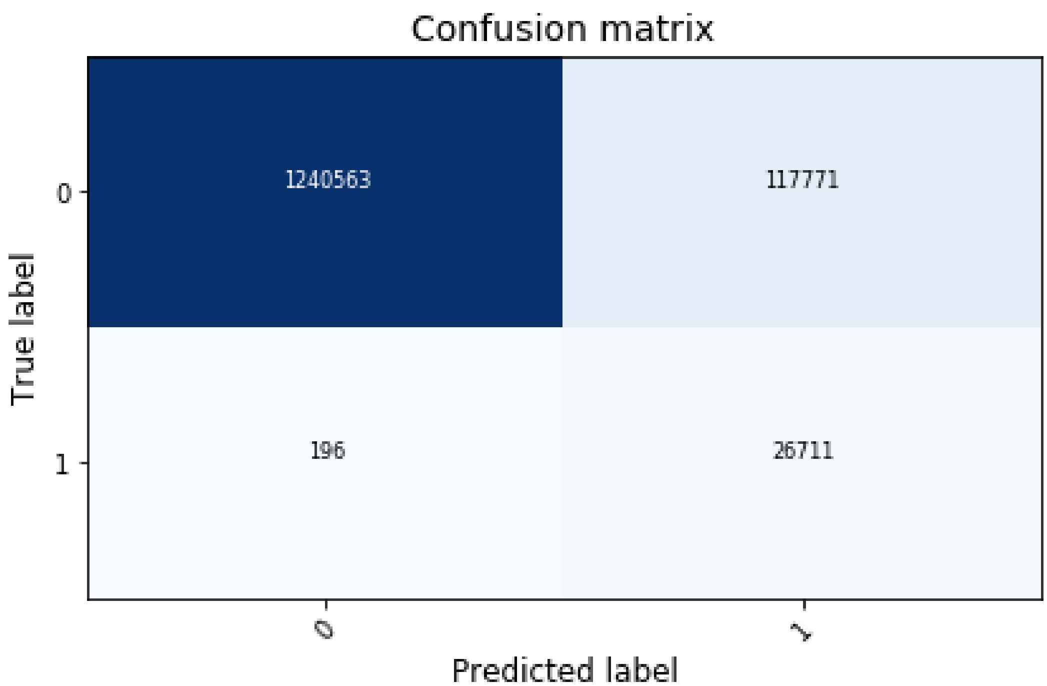
Now, that was a decisiontree model - lets try logistic regression as a classifier for predicting cancellation. Just like last time, wet'll create the label and featurevector using RFormula and create training- and test splits before training the model.

```
In [67]: # https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
         print(str(datetime.datetime.now())+": Generating feature-vector")
         formula = RFormula(
             #featuresCol="features",
             formula="Cancelled ~ DepDelay + DepTime + Distance + DayOfWeek",
             labelCol="label")

         outputForLogReg = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('None')).\
```

```python
                  transform(flightsWithDelayStatus.na.fill(0).na.fill('None')).\
                  select("features","label")

#output.limit(10).toPandas()
(trainingDataForLogReg, testDataForLogReg) = \
outputForLogReg.randomSplit([0.8,0.2],\
                            seed = 13234)

trainingDataForLogReg.groupBy(col('Label')).count().show()

print(str(datetime.datetime.now())+": Downsampling majority class")
trainingDataForLogRegBalanced = trainingDataForLogReg.where(col('label')==0).\
                                sample(False, (108016/5442478), 42).\
                                unionAll(trainingDataForLogReg.\
                                         where(col('label')==1))

trainingDataForLogRegBalanced.groupBy(col('Label')).count().show()

from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
print(str(datetime.datetime.now())+": Fitting model")
lrModel = lr.fit(trainingDataForLogRegBalanced)

# Predict, using the model
print(str(datetime.datetime.now())+": Predicting... and computing metrics")
predictions = lrModel.transform(testDataForLogReg)
predictions.groupby("prediction").count().toPandas()

print(str(datetime.datetime.now())+": Building pretty confusion matrix")
true=predictions.select('label').toPandas() #Serializing to native Python (Pandas)
                                            # dataframe

predicted=predictions.select('prediction').toPandas()

pretty_print_conf_matrix(true,\
                         predicted,\
                         classes=[0,1],\
                         normalize=False,\
                         title='Confusion matrix',\
                         cmap=plt.cm.Blues)
```

```
2018-06-10 17:47:58.474928: Generating feature-vector
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|5444514|
|  1.0| 108360|
+-----+-------+
```

```
2018-06-10 17:48:05.106119: Downsampling majority class
+-----+------+
|Label| count|
+-----+------+
|  0.0|108493|
|  1.0|108360|
+-----+------+

2018-06-10 17:48:16.538772: Fitting model
2018-06-10 17:48:29.024122: Predicting... and computing metrics
2018-06-10 17:48:35.755952: Building pretty confusion matrix
```



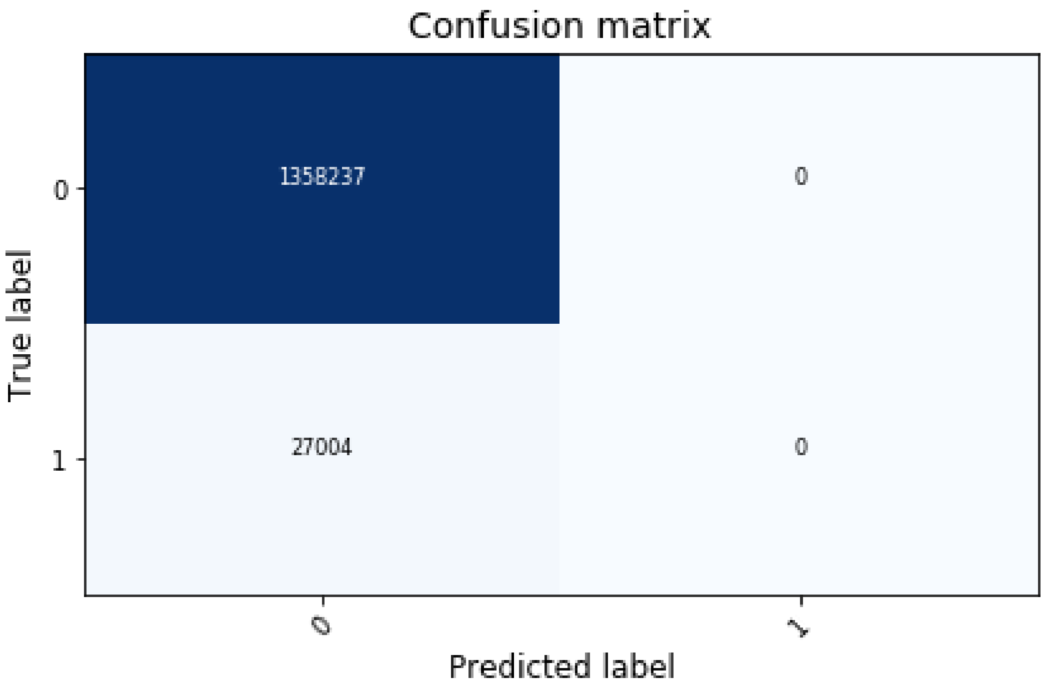
	precision	recall	f1-score	N Obs
0.0	1.00	0.91	0.95	1358334
1.0	0.18	0.99	0.31	26907
avg	0.98	0.91	0.94	1385241

Well, the precision when predicting fligths as cancled is not very good, so the feature-vector that worked for predicting delays, does not work for predicting cancellations - let go back to weather-features to see if that helps.

```
In [68]: # https://spark.apache.org/docs/2.2.0/ml-features.html#rformula
         print(str(datetime.datetime.now())+": Generating feature-vector")
         formula = RFormula(
             #featuresCol="features",
         formula="Cancelled ~ Tmax + TmaxFlag + Tmin + "+\
                 "TminFlag + Tavg + Depart + DewPoint + "+\
                 "WetBulb + Heat + Cool + Sunrise + Sunset + "+\
                 "CodeSum  + Depth + SnowFall + SnowFallFlag + "+\
                 "PrecipTotal + PrecipTotalFlag + StnPressure + "+\
                 "SeaLevel + ResultSpeed + ResultDir + "+\
                 "AvgSpeed + Max5Speed + Max5SpeedFlag + Max5Dir + "+\
                 "Max2Speed + Max2SpeedFlag + Max2Dir",\
             labelCol="label")

         outputForLogReg = formula.fit(flightsWithDelayStatus.na.fill(0).na.fill('None')).\
                           transform(flightsWithDelayStatus.na.fill(0).na.fill('None')).\
                           select("features","label")

         #output.limit(10).toPandas()
         (trainingDataForLogReg, testDataForLogReg) = outputForLogReg.\
         randomSplit([0.8,0.2],\
                      seed = 13234 )

         trainingDataForLogReg.groupBy(col('Label')).count().show()
         print(str(datetime.datetime.now())+": Downsampling majority class")

         trainingDataForLogRegBalanced = trainingDataForLogReg.where(col('label')==0).\
                                             sample(False,\
                                                    (108016/5442478),ú
                                                    42).\
                                         unionAll(trainingDataForLogReg.where(col('label')==1)

         trainingDataForLogRegBalanced.groupBy(col('Label')).count().show()

         from pyspark.ml.classification import LogisticRegression
         lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
         print(str(datetime.datetime.now())+": Fitting model")
         lrModel = lr.fit(trainingDataForLogRegBalanced)

         # Predict, using the model
         print(str(datetime.datetime.now())+": Predicting... and computing metrics")
         predictions = lrModel.transform(testDataForLogReg)
         predictions.groupby("prediction").count().toPandas()
```

```python
    print(str(datetime.datetime.now())+": Building pretty confusion matrix")
    true=predictions.select('label').toPandas() #Serializing to native Python (Pandas)
                                                # dataframe
    predicted=predictions.select('prediction').toPandas()
    pretty_print_conf_matrix(true,\
                             predicted,\
                             classes=[0,1],\
                             normalize=False,\
                             title='Confusion matrix',\
                             cmap=plt.cm.Blues)
```

```
2018-06-10 17:49:18.837260: Generating feature-vector
+-----+-------+
|Label|  count|
+-----+-------+
|  0.0|5444611|
|  1.0| 108263|
+-----+-------+

2018-06-10 17:50:14.042756: Downsampling majority class
+-----+------+
|Label| count|
+-----+------+
|  0.0|108496|
|  1.0|108263|
+-----+------+

2018-06-10 17:51:25.067593: Fitting model
2018-06-10 17:52:38.316585: Predicting... and computing metrics
2018-06-10 17:53:14.837631: Building pretty confusion matrix


/home/ubuntu/.local/lib/python3.5/site-packages/sklearn/metrics/classification.py:1135: Undefi
  'precision', 'predicted', average, warn_for)
```



	precision	recall	f1-score	N Obs
0.0	0.98	1.00	0.99	1358237
1.0	0.00	0.00	0.00	27004
avg	0.96	0.98	0.97	1385241

That was even worse - it seems that predicting cancellations from the present data is difficult. Now, multiple algorithms could be tried on this problem, and hyperparameter tuning could be applied as we did with the decision tree - this might, combined with some thorough feature analysis and selection imprrove the result.