|  |
| --- |
| PREDICTIONS du RETARD D’UN VOL  LIE AUX CONDITIONS CLIMATIQUES  MANIPULATION DE DONNEES MASSIVES |
|  |



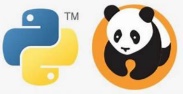
IASD 2019 / 2020 – Promo 1

CHANG Hy-Boui OMOLA Daniel PAN Frédéric

|  |
| --- |
| Table des matières  [1 Introduction 4](#_Toc56300624)  [1.1 Présentation du sujet 4](#_Toc56300625)  [1.2 La publication 5](#_Toc56300626)  [1.3 Méthodologie 6](#_Toc56300627)  [2 Les données sources et leur préparation 7](#_Toc56300628)  [2.1 Récupération des données 7](#_Toc56300629)  [**2.1.1** **Les données vol** 7](#_Toc56300630)  [**2.1.2** **Les données météo** 8](#_Toc56300631)  [2.2 Lien entre les données vol et données météo 11](#_Toc56300632)  [2.3 Nettoyage des données 13](#_Toc56300633)  [**2.3.1** **Liste des nettoyages** 13](#_Toc56300634)  [**2.3.2** **Programme de nettoyage sous PANDAS** 15](#_Toc56300635)  [**2.3.3** **Gain en tailles des données en sortie** 20](#_Toc56300636)  [3 Exploration des données 21](#_Toc56300637)  [4 Jointures 25](#_Toc56300638)  [**4.1.1** **Matériels** 25](#_Toc56300639)  [**4.1.2** **Programme scala** 25](#_Toc56300640)  [5 Pipeline et modèles 38](#_Toc56300641)  [5.1 Le programme pour le pipeline et le modèle à entrainer 38](#_Toc56300642)  [**5.1.1** **Préparation des données avant Pipeline** 38](#_Toc56300643)  [**5.1.2** **Pipeline** 43](#_Toc56300644)  [**5.1.3** **Les hyperparamètres et l’évaluation du modèle** 51](#_Toc56300645)  [5.2 Lancement du programme en traitement de données massives via le cluster Dauphine ou AWS 53](#_Toc56300646)  [**5.2.1** **Création du .jar** 53](#_Toc56300647)  [**5.2.2** **Connection au cluster Dauphine** 55](#_Toc56300648)  [Vérification des versions Spark et Scala du cluster Dauphine 58](#_Toc56300649)  [**5.2.3** **Chargement des fichiers (sources et .jar) dans HDFS** 58](#_Toc56300650)  [**5.2.4** **Lancement des job Spark en local, Stand Alone et sur Yarn** 60](#_Toc56300651)  [**5.2.5** **Tentatives d’optimisation** 61](#_Toc56300652)  [**5.2.6** **Conclusion sur les clusters Dauphine et AWS :** 62](#_Toc56300653)  [5.3 Traitement de données massives via DATABRICKS 64](#_Toc56300654)  [6 Résultats 67](#_Toc56300655)  [6.1 Résultats sur les deux versions de datasets vol que nous avons testé 67](#_Toc56300656)  [**6.1.1** **Résultats avec le premier dataset1 de vol** 67](#_Toc56300657)  [**6.1.2** **Résultats avec le second dataset2 de vol** 67](#_Toc56300658)  [6.2 Interprétation des résultats 68](#_Toc56300659)  [6.3 Pour aller plus loin … 70](#_Toc56300660)  [7 Conclusions sur le travail demandé 72](#_Toc56300661)  [7.1 Les difficultés rencontrées 72](#_Toc56300662)  [**7.1.1** **Incertitudes quant à l’architecture précise du cluster Dauphine** 72](#_Toc56300663)  [**7.1.2** **Organisation et gestion de projets** 72](#_Toc56300664)  [**7.1.3** **Temps d’exécution global (plusieurs dizaines heures voire de jours)** 73](#_Toc56300665)  [**7.1.4** **Log d’erreur Spark peu ou pas explicite** 74](#_Toc56300666)  [7.2 Ce que nous avons appris 74](#_Toc56300667)  [Annexe 1a – Création des .JAR avec SBT sous Linux 76](#_Toc56300668)  [Annexe 1b – Création des .JAR avec IntelliJ sous Windows 78](#_Toc56300669) |

|  |
| --- |
| IntroductionPrésentation du sujet Dans le domaine du transport en commun, les délais et surtout le respect des horaires sont des composantes importantes pour mesurer la performance d’un système. Les compagnies aériennes, ferroviaires ou routières ont des engagements sur le respect des délais.  Nous allons dans ce projet nous intéresser tout particulièrement aux délais aériens.  Le moindre retard a des conséquences qui peuvent être catastrophiques :   * Tout d’abord pour le voyageur qui peut rater sa correspondance, dont le nombre de jours en congé est limité, qui doit participer à une réunion d’affaire importante, qui doit aller visiter un malade, etc. * Ensuite pour la compagnie aérienne qui doit gérer les réclamations des voyageurs, trouver des solutions alternatives pour satisfaire les passagers (souvent coûteuses), mais aussi gérer les retards en cascade causés par le premier retard. D’un point de vue économique, une estimation situerait un surcoût de 33 milliards US $ par an pour les retards aériens aux USA.   Les causes de retard sont multiples :   * L’avion n’est pas arrivé dans les temps et il est nécessaire d’effectuer un certain nombre d’actions pour préparer le vol suivant (briefing pilotes, acheminement des plateaux repas, chargement des bagages, prise de connaissance de l’état technique de l’appareil, carburant, entrées des données de vol…) ; * L’avion prend du retard pour contourner une météo défavorable sur son trajet * L’avion subit un check technique complet et il y a des points de sécurité à effectuer ; * L’aéroport doit effectuer des contrôles sécurités sur les voyageurs ; * L’aéroport est en alerte sécurité, le personnel est en grève ; * Quelques voyageurs retardent le décollage de l’avion au départ (retard du voyageur, santé du voyageur, …) ; * L’avion est détourné au cours de son vol ; * Le vol est annulé.   L’étude qui suit va s’attacher à prédire les vols retardés en raison de conditions météo défavorables. Pour ce faire, nous entrainnerons un classifieur de type Random Forest sur les données de vols et météos couvrants l’année 2012 et 2013.  Compte tenu du caractère massif des données (1.7\*24 = 41 GO pour les vols, 6\*24 = 144 GO pour les conditions météo) et de la nécéssité de réaliser des traitements distribués, qui en découle, le framework Spark sera grandement privilégié.   La publication Nous nous inspirons de cette publication pour la préparation des données et mener les analyses et calculs adéquats :  Afin de mener à bien notre étude, nous nous appuierons sur la publication **« Using Scalable Data Mining for Predicting Flight Delays »** (cf. détail ci-dessous).Cette dernière nous serautile autant pour la préparation et l’analyse des données, que pour l’implémentation du Classifier.   * Auteurs de l’université de Calabria : L. Belcastra, Dominico Taila, Fabrizio Marozzo, Paolo Trunfio * Date de parution : janvier 2016 * Lieu de parution : ACM Transactions on Intelligent Systems and Technology   La publication est :   * Téléchargeable à l’adresse suivante : <https://www.coursehero.com/file/63005424/TIST-Flight-Delay-finalpdf/> * Voici la version pdf :    Méthodologie Les différentes étapes du projet sont les suivants :   * Etape 1 : Récupération des données et nettoyage ; * Etape 2 : Exploration des données ; * Etape 3 : Jointure entre les données de vol et météo ; * Etape 4 : Traitement des données, pipeline ; * Etape 5 : Résultats.   Les langages utilisés :  Les langages de programmation utilisés pour nos traitements sont Python et Scala.  Pour les deux premières étapes (récupération-nettoyage et exploration des données) le language python a été privilégié et notamment la librairie PANDAS.  Pour les étapes suivantes (jointures, pipeline, entraînement du modèle) où le traitement distribué s’impose), le langage Scala a été favorisé.  Les clusters utilisés :  Traitement données massives sur plusieurs clusters : Dauphine, AWS, Databricks. |







# Les données sources et leur préparation

Cette première partie de préparation des données (compréhension des champs, nettoyage des valeurs inutilisables, complétude des informations, etc) mérite que nous y passions un certain temps pour pouvoir aborder la suite du projet sereinement.

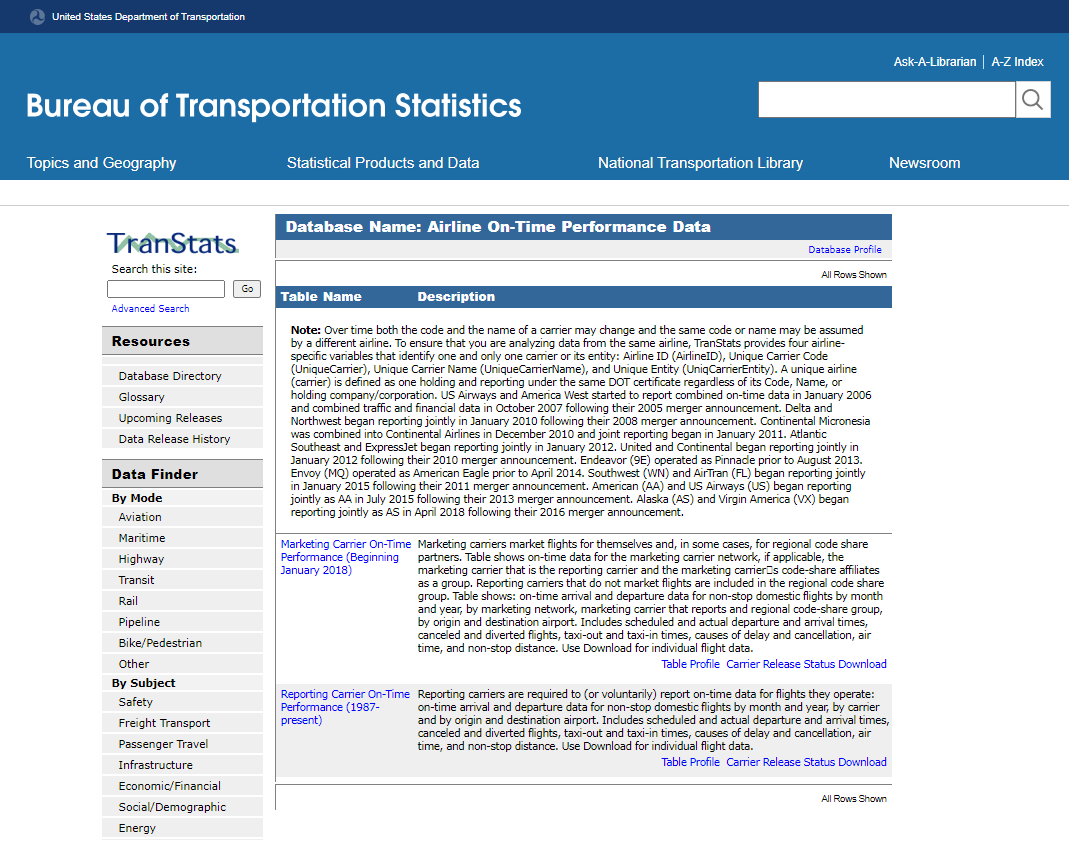
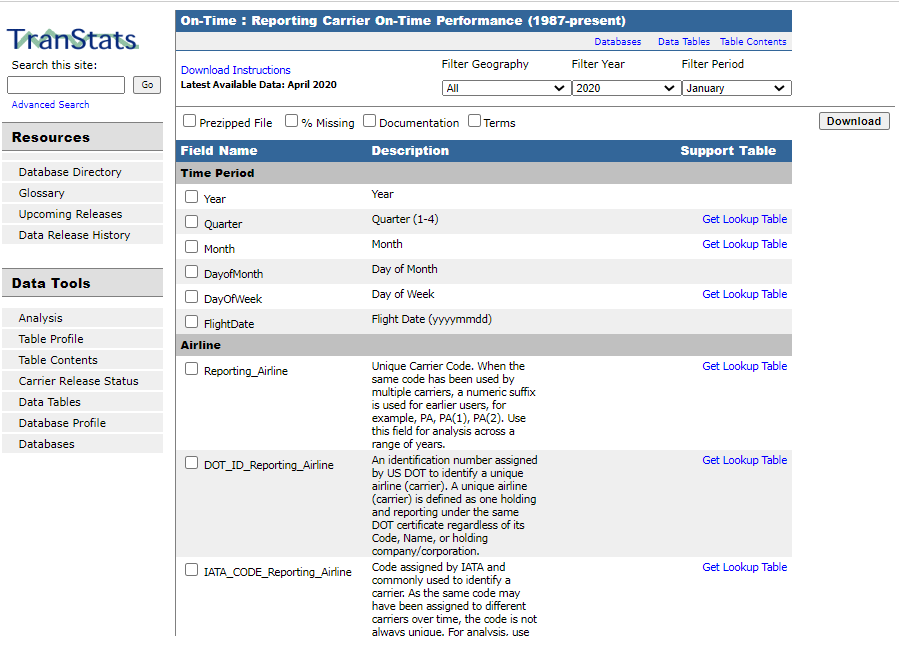


## Récupération des données

### **Les données vol**

Les données sur les vols sont disponibles et téléchargeables sur le site du « Bureau of Transportation Statistics ». Voici l’adresse du site :

<https://www.transtats.bts.gov/Tables.asp?DB_ID=120&DB_Name=Airline%20On-Time%20Performance%20Data&DB_Short_Name=On-Time>



Le site recense des informations récoltées depuis 1987.

Après une navigation sur le site pour comprendre son fonctionnement, nous avons sélectionné les champs qui sont utiles pour notre projet et nous avons commencé le téléchargement des données sur les années 2012 et 2013, mois par mois.

Pour information, 1 mois de données représente environ 600 000 lignes.

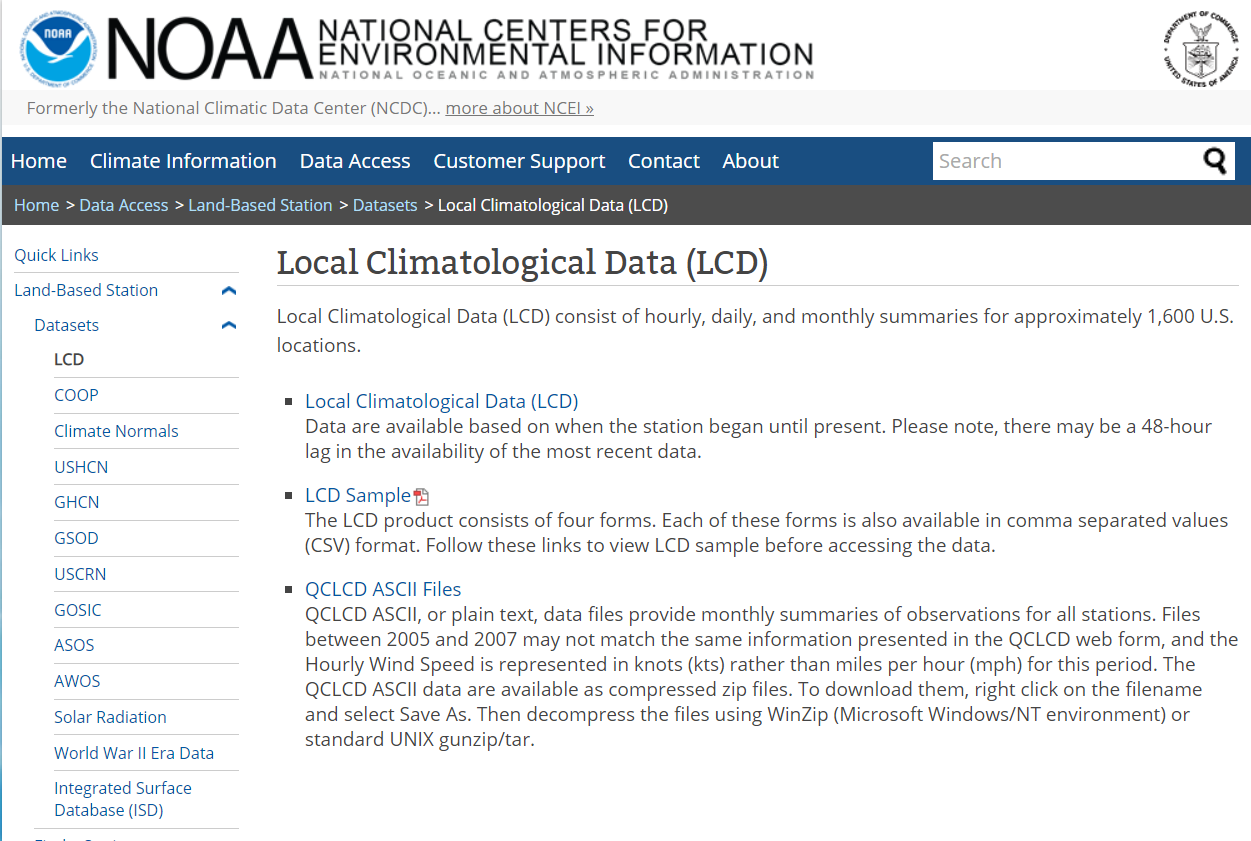
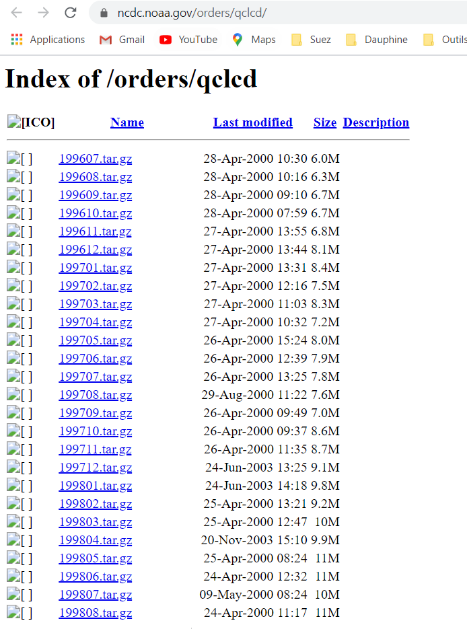
Voici les champs que nous avons choisis :

* **Time period** : Year, Quarter, Month, DayOfMonth, DayOfWeek, FlightDate
* **Airline** : Reporting\_Airline, DOT\_ID\_Reporting\_Airline, IATA\_CODE\_Reporting\_Airline, Tail\_Number, Flight\_Number\_Reporting\_Airline
* **Origin** : Origin Airport ID, Origin Airpot SeqID, Origin, OriginCityName, OriginState, OriginStateT ips, OriginStateName, OriginWac
* **Destination** : Destination Airport ID, Destination Airpot SeqID, Destination, DestinationCityName, DestinationState, DestinationStateTips, DestinationStateName, DestinationWac
* **Departure Performance** : CRSDepTime, DepTime (actual departure local time, 4 chiffres), DepDelay, DepDelayMinutes, DepDel15 (si sup 15 min, alors 1), DepartureDelayGroups
* **Arrival Performance** : CRS Arrival Time, Actual Arrival Time, ArrDelay, ArrDelayMinutes, ArrDel15, ArrDel15, ArrivalDelayGroups
* **Cancellations and Diversions** : Cancelled (1=yes), Diverted (1=yes)
* **Flight Summaries** : CRS Elapsed Time, Actual Elapsed time, AirTime, Flights (numbers of flights), Distance (miles)
* **Cause of Delay** : CarrierDelay (en min), WeatherDelay (en min), NASDelay, SecurityDelay, LateAircraftDelay

### **Les données météo**

Les données sur les conditions météo sont disponibles sur le site du « National Center for Environmental information » :

<https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/quality-controlled-local-climatological-data-qclcd>



A nouveau, chaque mois se présente sous forme d’un fichier à part. Pour les années 2012 et 2013, nous devons télécharger des fichiers zippés, mois par mois (par exemple pour mars 2012 : QCLCD201203.zip).

Dans ce fichier zippé, nous ne gardons que 2 fichiers : 201203hourly.txt et 201203station.txt.

C’est le fichier AAAAMMhourly.txt qui nous intéresse, car il renferme les relevés météorologiques tout au long de la journée.

Nous gardons également le fichier AAAAMMstation.txt car il va nous permettre de faire le lien entre les données vol et les données météo.

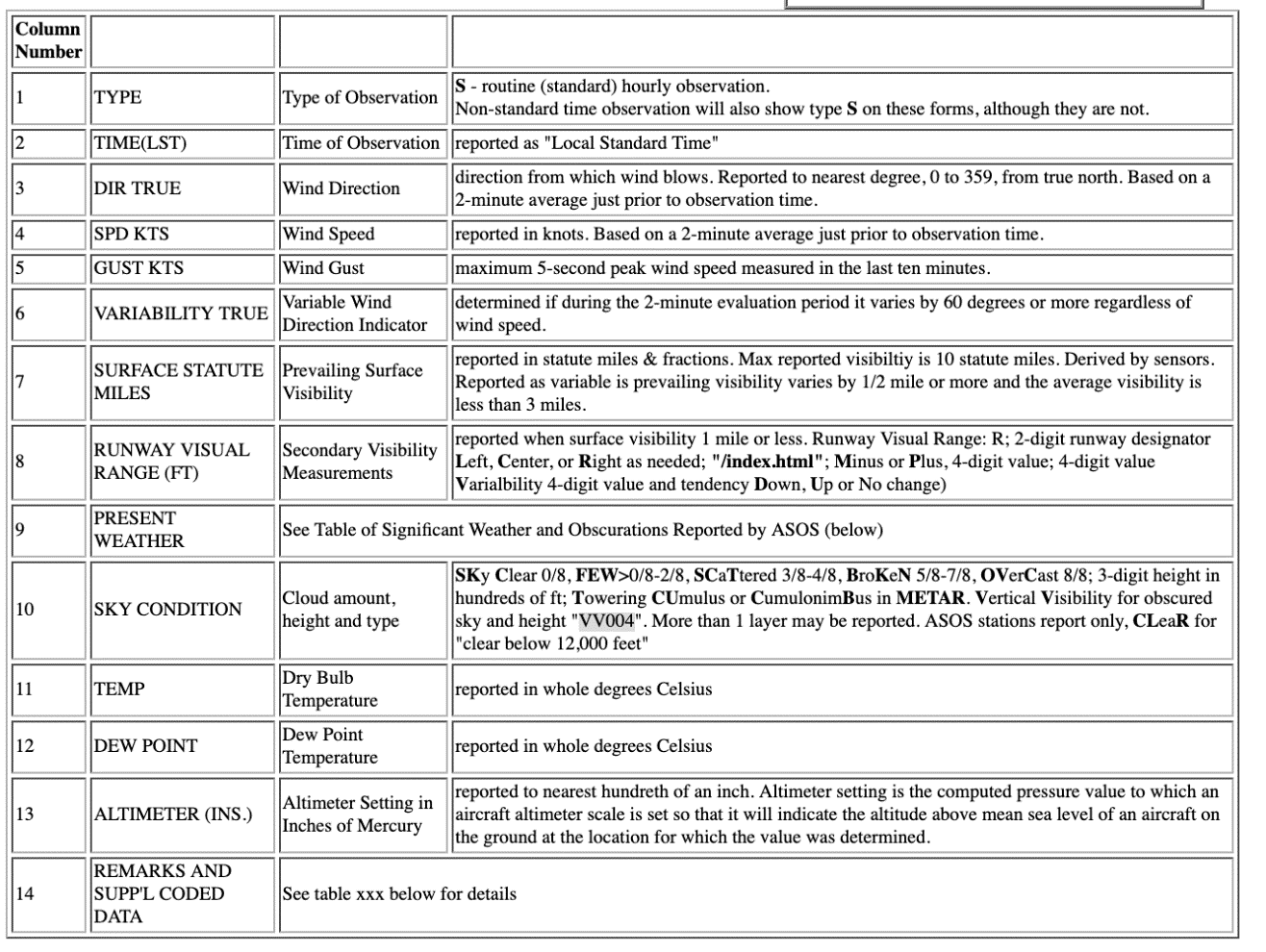
Voici les champs du fichier AAAAMMhourly.txt  que nous avons récupérés:

* WBAN
* Date, Time
* StationType
* SkyCondition, SkyConditionFlag
* Visibility, VisibilityFlag
* WeatherType, WeatherTypeFlag
* DryBulbFarenheit, DryBulbFarenheitFlag, DryBulbCelsius, DryBulbCelsiusFlag
* WetBulbFarenheit, WetBulbFarenheitFlag, WetBulbCelsius, WetBulbCelsiusFlag
* DewPointFarenheit, DewPointFarenheitFlag, DewPointCelsius, DewPointCelsiusFlag
* RelativeHumidity, RelativeHumidityFlag
* WindSpeed, WindSpeedFlag
* WindDirection, WindDirectionFlag
* ValueForWindCharactere, ValueForWindCharacterFlag
* StationPressure, StationPressureFlag
* PressureTendency, PressureTendencyFlag
* PressureChange, PressureChangeFlag
* SeaLevelPressure, SeaLevelPressureFlag
* RecordType, RecordTypeFlag
* HourlyPrecip, HourlyPrecipFlag
* Altimeter, AltimeterFlag

Le détail des champs est consultable à l’adresse suivante :

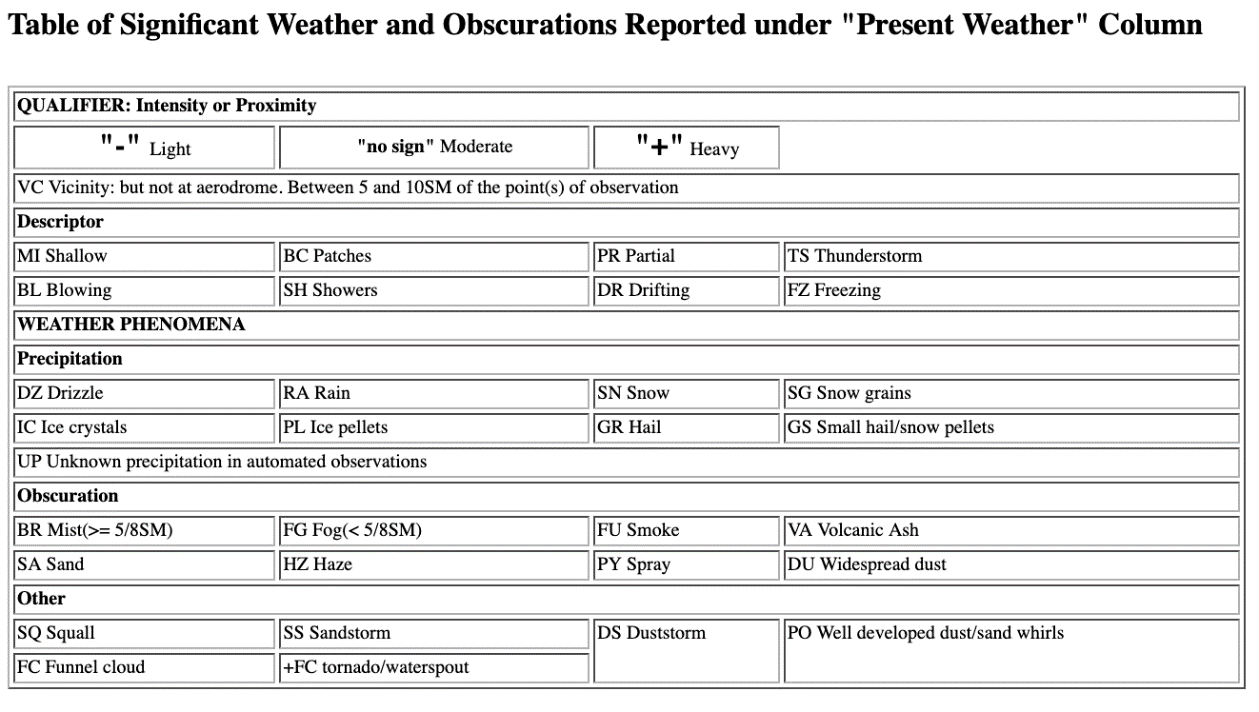
<https://www.ncdc.noaa.gov/orders/qclcd/ExplanationofCodes.htm>

En voici les principales informations :



On pourra remarquer que pour le champ « Sky condition », les informations les plus importantes semblent être :

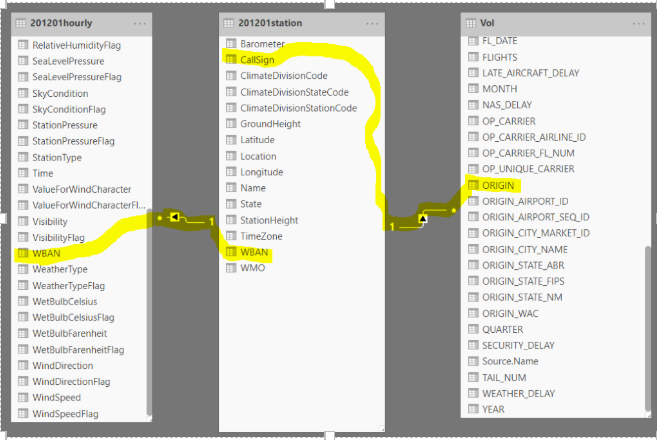
* celles contenant le préfixe “VV” pour Vertical Visibility suivi d’une valeur numérique sur 4 caractères
* et celles contenant la valeur “CLR” pour Clear under 12,000 feets.



## Lien entre les données vol et données météo

Maintenant que nous avons choisi les données utiles et récupéré les données de vol et météorologiques, il nous faut établir des correspondances nécessaires à la réalisation de jointures.

* Dans les données de vols, l’aéroport est identifié sous 3 lettres.
* Dans les données météorologiques (hourly.txt), les lieux sont repérés à l’aide d’un code WBAN à 5 chiffres.
* Nous utilisons les informations disponibles dans le fichier AAAAMMstation.txt pour obtenir la correspondance entre l’aéroport (3 lettres) et la localisation de la station météo (5 chiffres).





Nous générons une table pivot à partir des valeurs uniques WBAN trouvées dans les fichiers AAAAMMstation.txt des années 2012 et 2013.

Voici une copie de notre table pivot :



|  |  |  |
| --- | --- | --- |
| AIRPORT | AIRPORT\_STATE | wban |
| ABE | PA | 14737 |
| ABI | TX | 13962 |
| ABQ | NM | 23050 |
| ABR | SD | 14929 |
| ABY | GA | 13869 |
| ACK | MA | 14756 |
| ACT | TX | 13959 |
| … | … | … |

## Nettoyage des données

Précision pour la suite : Notre analyse est **uniquement sur les vols qui atterrissent en retard** à l’aéroport d’arrivée (et non ceux qui ont décollé en retard).

### **Liste des nettoyages**

Dans la publication sur lequel nous basons notre projet, le nettoyage préalable des données est précisé comme suit :

Pour les données de vols :

**Notion de vols en retard :**

Les résultats sont à mettre en regard de la méthodologie appliquée. Notamment la notion de « vols en retard ».

**Hypothèse D1** : Les vols en retards sont ceux avec le champ **ARRDEL15=1**. C’est-à-dire lorsque le cumul des retards pour différentes raisons (mise à disposition tardive, sécurité, etc.) dépasse les 15 minutes.

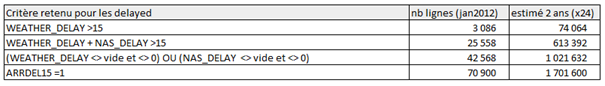
**Hypothèse D2** : Les vols ayant comme critère **NAS\_DELAY + WEATHER\_DELAY> 15**.

**Hypothèse D3** : Les vols ayant comme critère **WEATHER\_DELAY > 15**.

Ces hypothèses influent directement sur :

* La quantité de données (après rebalancing 50% - 50%) ;
* L’homogénéité des données ;
* **Le temps de calcul nécessaire, les ressources utilisées et les problèmes techniques susceptibles d’être rencontrés,**
* **Les résultats qui en découlent, leur comparabilité et la généralisation possible.**

Voici une illustration :



En termes de volumétrie de données, si on compare la quantité de données entre le 1er et le 4ème critère, il y a un facteur multiplicateur de 23.

* Les vols pour lesquels nous ne disposons pas de relevés météorologiques sont à supprimer (pas de correspondance avec une station météo).
* Les vols annulés ou déviés ne font pas parti des données d’entrée non plus. Il s’agit des lignes de données vol qui sont indiquées "Cancelled = 1" ou "Diverted = 1".

Nous avons essayé le programme avec 2 données VOL filtrées différemment :

Dataset1 (hypothèse **D1**) :

* Nous souhaitons prédire les retards > à 15 min, alors, il faut prendre le champ "ArrDel15" comme label. Cette information est donc à conserver.

Dataset2 (hypothèse **D3**) :

* Nous souhaitons prédire les retards uniquement liés à la météo > à 15 min, alors, il faut prendre le champ "WeatherDelay" comme label. Cette information est donc à conserver.

Pour les données METEO :

* Toutes les lignes de données qui ne correspondent pas à un aéroport, correspondent à des lignes inutiles. Cela nous permet de réduire la taille des fichiers sources à traiter.

### **Programme de nettoyage sous PANDAS**

Voici le programme de nettoyage :



Importation des librairies utiles et mise en place de la localisation des fichiers :

|  |
| --- |
| import pandas as pd  from datetime import datetime  from datetime import timedelta  path = "S:/Desktop/Flight Project/Data/" |

Nettoyage des données VOL

|  |
| --- |
| def MEF\_time\_vol(time\_t):  final =""  if len(time\_t) == 4:  if str(time\_t)[:2]=='24': # correction des Deptime= 2400  final = "00:" + str(time\_t)[2:4] + ":00"  else :  final = str(time\_t)[:2] + ":" + str(time\_t)[2:4] + ":00"  if len(time\_t) == 3:  final = "0" + str(time\_t)[:1] + ":" + str(time\_t)[1:3] + ":00"  if len(time\_t) == 2:  final = "00" + ":" + str(time\_t) + ":00"  if len(time\_t) == 1:  final = "00:0" + str(time\_t) + ":00"  if len(time\_t) == 0:  final = "00:00:00"  return final  # notre propre critère de vols en retard dûs à la météo, on recalcule "ARR\_DEL15"  def update\_arr\_delay(temp\_vol):  #temp\_vol['ARR\_DEL15\_OLD'] = temp\_vol['ARR\_DEL15']  temp\_vol['ARR\_DEL15'] = ((temp\_vol['WEATHER\_DELAY'] >= 15).astype(int))  return(temp\_vol)  def rebalance(temp\_vol):  col = ['YEAR', 'QUARTER', 'MONTH', 'DAY\_OF\_MONTH', 'DAY\_OF\_WEEK',  'FL\_DATE', 'OP\_UNIQUE\_CARRIER', 'OP\_CARRIER\_AIRLINE\_ID',  'OP\_CARRIER', 'TAIL\_NUM', 'OP\_CARRIER\_FL\_NUM', 'ORIGIN\_AIRPORT\_ID',  'ORIGIN\_AIRPORT\_SEQ\_ID', 'ORIGIN\_CITY\_MARKET\_ID', 'ORIGIN',  'ORIGIN\_CITY\_NAME', 'ORIGIN\_STATE\_ABR', 'ORIGIN\_STATE\_FIPS',  'ORIGIN\_STATE\_NM', 'ORIGIN\_WAC', 'DEST\_AIRPORT\_ID',  'DEST\_AIRPORT\_SEQ\_ID', 'DEST\_CITY\_MARKET\_ID', 'DEST',  'DEST\_CITY\_NAME', 'DEST\_STATE\_ABR', 'DEST\_STATE\_FIPS',  'DEST\_STATE\_NM', 'DEST\_WAC', 'CRS\_DEP\_TIME', 'DEP\_TIME',  'DEP\_DELAY', 'DEP\_DELAY\_NEW', 'DEP\_DEL15', 'DEP\_DELAY\_GROUP',  'CRS\_ARR\_TIME', 'ARR\_TIME', 'ARR\_DELAY', 'ARR\_DELAY\_NEW',  'ARR\_DEL15', 'ARR\_DELAY\_GROUP', 'CANCELLED', 'DIVERTED',  'CRS\_ELAPSED\_TIME', 'ACTUAL\_ELAPSED\_TIME', 'AIR\_TIME', 'FLIGHTS',  'DISTANCE', 'CARRIER\_DELAY', 'WEATHER\_DELAY', 'NAS\_DELAY',  'SECURITY\_DELAY', 'LATE\_AIRCRAFT\_DELAY', 'Unnamed: 53']  temp\_vol\_rebalanced = pd.DataFrame(columns=col)  delayed\_flight= temp\_vol[temp\_vol.ARR\_DEL15==1]  index\_delayed\_flight = temp\_vol[temp\_vol.ARR\_DEL15==1].index  #print(" - index\_delayed\_flight : \n\t %s"%str(index\_delayed\_flight))  nb\_delayed\_flight = len(delayed\_flight)  #print(" - nb\_delayed\_flight : %d"%nb\_delayed\_flight)  temp\_vol = temp\_vol.sample(frac=1)  ontime\_flight = temp\_vol[~temp\_vol.index.isin(index\_delayed\_flight)][:nb\_delayed\_flight]  #print(" - ontime\_flight : %d"%len(ontime\_flight))  temp\_vol\_rebalanced = temp\_vol\_rebalanced.append(delayed\_flight).append(ontime\_flight)  #temp\_vol\_rebalanced = temp\_vol\_rebalanced.sample(frac=1)  temp\_vol\_rebalanced['new\_date']= pd.to\_datetime(temp\_vol\_rebalanced['FL\_DATE'])  temp\_vol\_rebalanced = temp\_vol\_rebalanced.sort\_values(by=['new\_date','DEP\_TIME'],ascending = [True,True])  temp\_vol\_rebalanced = temp\_vol\_rebalanced.drop(columns = ['new\_date'])  return(temp\_vol\_rebalanced)  def get\_filtered\_monthly\_flight(df\_vol): # On exclut les diverted et les cancelled  temp\_vol = update\_arr\_delay(df\_vol)  temp\_vol = temp\_vol[(df\_vol.CANCELLED == 0.0) & \  (df\_vol.DIVERTED == 0.0)]  temp\_vol = temp\_vol.drop(columns=['Unnamed: 53'])  # -------------- Rebalancing -----------------------  temp\_vol = rebalance(temp\_vol)  return(temp\_vol)  def get\_flight\_file\_list():  flight\_2012\_file\_list = ['2012\_0' + str(x) + '\_T\_ONTIME\_REPORTING.csv' \  if x <10 else '2012\_' + str(x) + '\_T\_ONTIME\_REPORTING.csv' \  for x in range(1,13)]  flight\_2013\_file\_list = ['2013\_0' + str(x) + '\_T\_ONTIME\_REPORTING.csv' \  if x <10 else '2013\_' + str(x) + '\_T\_ONTIME\_REPORTING.csv' \  for x in range(1,13)]  flight\_file\_list =[]  flight\_file\_list.extend(flight\_2012\_file\_list)  flight\_file\_list.extend(flight\_2013\_file\_list)  return(flight\_file\_list)  def get\_flight\_consolidated():  col = ['YEAR', 'QUARTER', 'MONTH', 'DAY\_OF\_MONTH', 'DAY\_OF\_WEEK',  'FL\_DATE', 'OP\_UNIQUE\_CARRIER', 'OP\_CARRIER\_AIRLINE\_ID',  'OP\_CARRIER', 'TAIL\_NUM', 'OP\_CARRIER\_FL\_NUM', 'ORIGIN\_AIRPORT\_ID',  'ORIGIN\_AIRPORT\_SEQ\_ID', 'ORIGIN\_CITY\_MARKET\_ID', 'ORIGIN',  'ORIGIN\_CITY\_NAME', 'ORIGIN\_STATE\_ABR', 'ORIGIN\_STATE\_FIPS',  'ORIGIN\_STATE\_NM', 'ORIGIN\_WAC', 'DEST\_AIRPORT\_ID',  'DEST\_AIRPORT\_SEQ\_ID', 'DEST\_CITY\_MARKET\_ID', 'DEST',  'DEST\_CITY\_NAME', 'DEST\_STATE\_ABR', 'DEST\_STATE\_FIPS',  'DEST\_STATE\_NM', 'DEST\_WAC', 'CRS\_DEP\_TIME', 'DEP\_TIME',  'DEP\_DELAY', 'DEP\_DELAY\_NEW', 'DEP\_DEL15', 'DEP\_DELAY\_GROUP',  'CRS\_ARR\_TIME', 'ARR\_TIME', 'ARR\_DELAY', 'ARR\_DELAY\_NEW',  'ARR\_DEL15', 'ARR\_DELAY\_GROUP', 'CANCELLED', 'DIVERTED',  'CRS\_ELAPSED\_TIME', 'ACTUAL\_ELAPSED\_TIME', 'AIR\_TIME', 'FLIGHTS',  'DISTANCE', 'CARRIER\_DELAY', 'WEATHER\_DELAY', 'NAS\_DELAY',  'SECURITY\_DELAY', 'LATE\_AIRCRAFT\_DELAY']  flight\_consolidated =pd.DataFrame(columns=col)  flight\_file\_list = get\_flight\_file\_list()#[:1]  for file\_ in flight\_file\_list :  df\_vol = pd.read\_csv(path + file\_)  data = get\_filtered\_monthly\_flight(df\_vol)  flight\_consolidated =flight\_consolidated.append(data)  return(flight\_consolidated)  flight\_24M = get\_flight\_consolidated()  flight\_24M.to\_csv(path + 'flight\_24M.csv',index = False)  flight\_24M = pd.read\_csv(path + 'flight\_24M.csv')  flight\_1M=flight\_24M[(flight\_24M.YEAR==2012)&(flight\_24M.MONTH==1)]  flight\_6M=flight\_24M[(flight\_24M.YEAR==2012)&(flight\_24M.MONTH<=6)]  flight\_12M=flight\_24M[(flight\_24M.YEAR==2012)&(flight\_24M.MONTH<=12)]  flight\_1M.to\_csv(path + 'flight\_1M.csv',index = False)  flight\_6M.to\_csv(path + 'flight\_6M.csv',index = False)  flight\_12M.to\_csv(path + 'flight\_12M.csv',index = False)  print(flight\_1M.shape,flight\_6M.shape,flight\_12M.shape,flight\_24M.shape)  #!/usr/bin/env python3  # -\*- coding: utf-8 -\*-  """  Created on Thu Apr 30 15:45:51 2020  @author: fredericpan  """  import pandas as pd  #import datetime  from datetime import datetime  from datetime import timedelta  #from datetime import date  def MEF\_time(time\_t):  final =""  if len(time\_t) == 4:  if str(time\_t)[:2]=='24': # correction des Deptime= 2400  final = "00:" + str(time\_t)[2:4] + ":00"  else :  final = str(time\_t)[:2] + ":" + str(time\_t)[2:4] + ":00"  if len(time\_t) == 3:  final = "0" + str(time\_t)[:1] + ":" + str(time\_t)[1:3] + ":00"  if len(time\_t) == 2:  final = "00" + ":" + str(time\_t) + ":00"  if len(time\_t) == 1:  final = "00:0" + str(time\_t) + ":00"  if len(time\_t) == 0:  final = "00:00:00"  return final  # Importation des données des vols  #path = "S:\Desktop\Flight Project\Data\Flight\_Clean\_Data"  files = ['flight\_1M.csv','flight\_6M.csv','flight\_12M.csv','flight\_24M.csv']  for fl in files:  df1 = pd.read\_csv(fl)  # On exclut les Diverted et les Cancelled  temp = df1  #temp = df1[df1['Cancelled'] == 0 ]  #temp = temp[temp['Diverted'] == 0]  # Concaténation des date & heure  # Tad : Actual departure time  temp['Tad']=temp.apply(lambda row: str(row.FL\_DATE) + " " + str(MEF\_time(str(int(row.DEP\_TIME)))) , axis = 1)  # Taa : Actual arrival time  temp['Taa']=temp.apply(lambda row: str(row.FL\_DATE) + " " + str(MEF\_time(str(int(row.ARR\_TIME)))) , axis = 1)  # recalcul des heures de départ et d'arrivés initiales  temp['Tsd']=temp.apply(lambda row: str(datetime.fromisoformat(row.Tad) + timedelta(minutes = -row.DEP\_DELAY)) , axis = 1)  temp['Tsa']=temp.apply(lambda row: str(datetime.fromisoformat(row.Taa) + timedelta(minutes = -row.ARR\_DELAY)) , axis = 1)  # Convertion de ArrDel15 en booléen  temp['Th']=temp.apply(lambda row: bool(row.ARR\_DEL15) , axis = 1)  # Sélection des champs en sortie  temp = temp[['ORIGIN','DEST','Tad','Tsd','Taa','Tsa','Th']]  # Exportation du CSV réduit à notre périmètre  temp.to\_csv (r'Clean/'+ fl, index = False, header=True)  print(temp.head()) |

Nettoyage des données METEO

|  |
| --- |
| def MEF\_time\_meteo(time\_t):  final =""  if len(time\_t) == 4:  final = str(time\_t)[:2] + ":" + str(time\_t)[2:4]+ ":00"  if len(time\_t) == 3:  final = "0" + str(time\_t)[:1] + ":" + str(time\_t)[1:3]+ ":00"  if len(time\_t) == 2:  final = "00" + ":" + str(time\_t)+ ":00"  if len(time\_t) == 1:  final = "00:0" + str(time\_t)+ ":00"  if len(time\_t) == 0:  final = "00:00:00"  return final  def get\_weather\_file\_list():  weather\_2012\_file\_list = ['20120' + str(x) + 'hourly.txt' \  if x <10 else '2012' + str(x) + 'hourly.txt' \  for x in range(1,13)]  weather\_2013\_file\_list = ['20130' + str(x) + 'hourly.txt' \  if x <10 else '2013' + str(x) + 'hourly.txt' \  for x in range(1,13)]  weather\_file\_list =[]  weather\_file\_list.extend(weather\_2012\_file\_list)  weather\_file\_list.extend(weather\_2013\_file\_list)  return(weather\_file\_list)  #weather\_file\_list = get\_weather\_file\_list()  #weather\_file\_list[:12]  def get\_weather\_monthly\_flight(df\_meteo,df\_pivot,airports):  # Jointure gauche sur les WBAN  # WBAN dans df\_meteo (hourly.txt)  # wban dans df\_pivot  df\_left\_merge = pd.merge(df\_meteo, df\_pivot, left\_on='WBAN', right\_on='wban',how= 'left')  #print(df\_left\_merge.head())  df\_left\_merge = df\_left\_merge[df\_left\_merge.AIRPORT.isin(airports)]  # sélection des données dans la liste des aéroports qui nous intéressent  temp\_met = df\_left\_merge[df\_left\_merge['AIRPORT'].notnull()]  # Mise en forme de la date en YYYY-MM-DD  temp\_met['Date2']=temp\_met.apply(lambda row: str(row.Date)[:4] + "-" + str(row.Date)[4:6] + "-" + str(row.Date)[6:8] , axis = 1)  # concaténation avec l'heure : YYY-MM-DD 0015  temp\_met['timest2']=temp\_met.apply(lambda row: str(row.Date2) + " " + str(MEF\_time\_meteo(str(row.Time))) , axis = 1)  # Sélection de données météo qui nous sont utiles  temp\_met = temp\_met[['timest2','AIRPORT','RelativeHumidity','WindDirection','WindSpeed','StationPressure','DewPointFarenheit','SkyCondition','WeatherType', 'Visibility']]  temp\_met.rename(columns = {'AIRPORT':'A','timest2':'t','DewPointFarenheit': 'T','RelativeHumidity':'H', 'WindDirection':'Wd', 'WindSpeed':'Ws', 'StationPressure':'P','SkyCondition':'S','Visibility':'V','WeatherType':'D'}, inplace = True)  return(temp\_met)  def get\_weather\_consolidated():  col = ['t', 'A', 'H', 'Wd', 'Ws', 'P', 'T', 'S', 'D', 'V']  weather\_consolidated =pd.DataFrame(columns=col)  weather\_file\_list = get\_weather\_file\_list()#[:1]  # Importation de la liste pivot wban-airport qui nous intéressent  #df\_pivot = pd.read\_csv('/content/drive/My Drive/Colab Notebooks/Flight/Table\_pivot\_OD\_WBAN.csv', sep=";")  df\_pivot = pd.read\_csv(path + 'Table\_pivot\_OD\_WBAN.csv', sep=";")  flight\_24M = pd.read\_csv(path + 'flight\_24M.csv')  airports = list(flight\_24M.ORIGIN.unique())  airports.extend(list(flight\_24M.DEST))  airports = list(set(airports))  for file\_ in weather\_file\_list :  # Importation des données méteo  df\_meteo = pd.read\_csv(path + file\_)  data = get\_weather\_monthly\_flight(df\_meteo,df\_pivot,airports)  weather\_consolidated = weather\_consolidated.append(data)  return(weather\_consolidated)  weather\_24M = get\_weather\_consolidated()  weather\_24M.to\_csv(path + 'weather\_24M.csv',index = False)  weather\_24M.shape  weather\_24M = pd.read\_csv(path + 'weather\_24M.csv')  weather\_24M['Y']= weather\_24M.t.apply(lambda t : int(t[:4]))  #weather\_24M['Y'].astype(int)  weather\_24M['M']= weather\_24M.t.apply(lambda t : int(t[5:7]))  #weather\_24M['M'].astype(int)  #weather\_24M.head()  weather\_1M=weather\_24M[(weather\_24M.Y==2012)&(weather\_24M.M==1)].drop(columns=['Y', 'M'])  weather\_6M=weather\_24M[(weather\_24M.Y==2012)&(weather\_24M.M<=6)].drop(columns=['Y', 'M'])  weather\_12M=weather\_24M[(weather\_24M.Y==2012)&(weather\_24M.M<=12)].drop(columns=['Y', 'M'])  weather\_1M.to\_csv(path + 'weather\_1M.csv',index = False)  weather\_6M.to\_csv(path + 'weather\_6M.csv',index = False)  weather\_12M.to\_csv(path + 'weather\_12M.csv',index = False)  print(weather\_1M.shape,weather\_6M.shape,weather\_12M.shape,weather\_24M.shape)  weather\_24M = pd.read\_csv(path + 'Flight\_Clean\_Data\_Sauvegarde\_0/weather\_24M.csv')  weather\_24M['t\_bis']= pd.to\_datetime(weather\_24M['t'][0][:10])  weather\_24M.shape  weather\_24M.head()  flight\_24M = pd.read\_csv(path + 'Flight\_Clean\_Data/flight\_24M.csv')  flight\_date = flight\_24M.FL\_DATE.unique().tolist()  len(flight\_date)  flight\_24M.head(25)  flight\_24M.shape  new\_weather = weather\_24M[weather\_24M.t\_bis.isin(flight\_date)]  new\_weather.shape  flight\_24M.shape |

Remarque :

Nous avons lancé ce programme de nettoyage des données sur les 24 mois de données de vols, et les 24 mois de données météorologiques :

* Les fichiers météo nettoyés sont nommés ainsi : OT\_AAAAMM.csv.
* Les fichiers vol nettoyés sont nommés ainsi : FT\_AAAAMM.csv.

### **Gain en tailles des données en sortie**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Données brutes | Données nettoyées avec vol dataset1 | Données nettoyées avec vol dataset2 |
| VOL | 1.7 GO \* 24 = 41 GO | 42 MO \* 24 = 1 GO | 15 MO \* 24 = 0.2 GO |
| METEO | 6 GO \* 24 = 144 GO | 442 MO \* 24 = 3 GO | 442 MO \* 24 = 3 GO |

Note : 1 GO = 1 000 MO = 1 000 000 KO

Nous avons donc considérablement réduit la taille de nos données. Mais cependant, cela n’est pas suffisant pour faire fonctionner le programme en local sur nos ordinateurs, il faudra utiliser les ressources en ligne.

# Exploration des données





Quelques statistiques :

|  |
| --- |
| df\_delay = df\_vol[df\_vol.ARR\_DEL15==1.00]  df\_nodelay = df\_vol[df\_vol.ARR\_DEL15==0.00]  delays = len(df\_delay)  nodelays = len(df\_nodelay)  airlines = sorted(df\_vol.OP\_UNIQUE\_CARRIER.unique())  origins = sorted(df\_vol.ORIGIN.unique(), reverse=True)  destinations = sorted(df\_vol.DEST.unique(), reverse=True)  numAirlines = len(airlines)  numOrigins = len(origins)  numDestinations = len(destinations)  minDist = df\_vol.DISTANCE.min()  maxDist = df\_vol.DISTANCE.max()  print("Sur des données de vol sur une période de 1 mois, nous avons :")  print("Nbre de vols en retard = %i" %delays)  print("Nbre de vol à l'heure = %i" %nodelays)  print("Nbre de compagnies aériennes = %i" %numAirlines)  print("Nbre d'aéroports de départ = %i" %numOrigins)  print("Nbre d'aéroports d'arrivée = %i" %numDestinations)  print("Distance minimum = %f" %minDist)  print("Distance maximum = %f" %maxDist) |

Nous obtenons les informations suivantes :

Sur des données de vol sur une période de 1 mois, nous avons :

Nbre de vols en retard = 70908

Nbre de vol à l'heure = 407117

Nbre de compagnies aériennes = 15

Nbre d'aéroports de départ = 287

Nbre d'aéroports d'arrivée = 287

Distance minimum = 31.000000

Distance maximum = 4963.000000

Le premier constat est qu'il y a davantage de vols à l'heure que ceux en retard (heureusement) ! Cependant, ce déséquilibre nous oblige à penser à ré-équilibrer les données avant le lancement de l'algorithme d'apprentissage.

import matplotlib.pyplot as plt

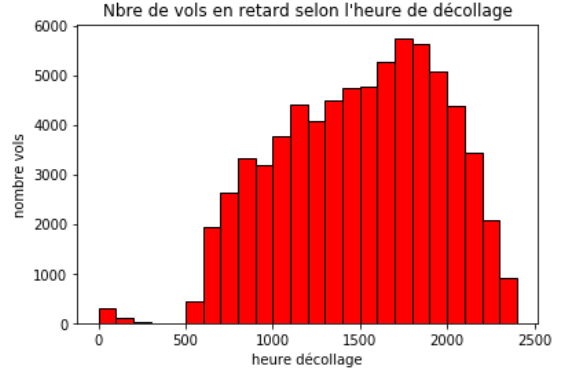
plt.hist(df\_delay.DEP\_TIME, range = (0, 2400), bins = 24, color = "red",

edgecolor = "black")

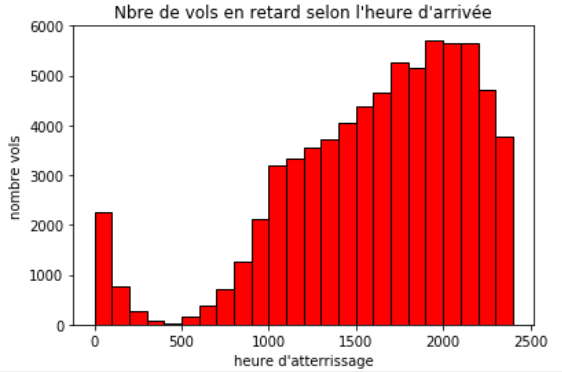
plt.xlabel("heure décollage")

plt.ylabel("nombre vols")

plt.title("Nbre de vols en retard selon l'heure de décollage")

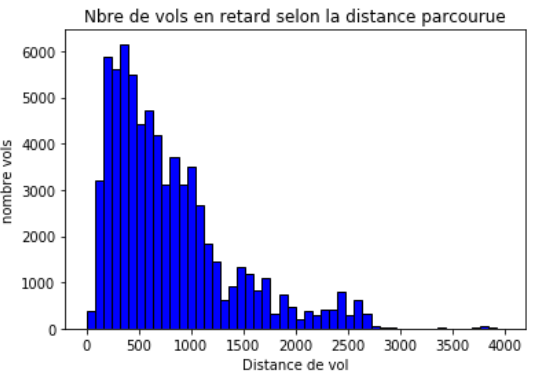


|  |
| --- |
| plt.hist(df\_delay.ARR\_TIME, range = (0, 2400), bins = 24, color = "red",  edgecolor = "black")  plt.xlabel("heure d'atterrissage")  plt.ylabel("nombre vols")  plt.title("Nbre de vols en retard selon l'heure d'arrivée") |



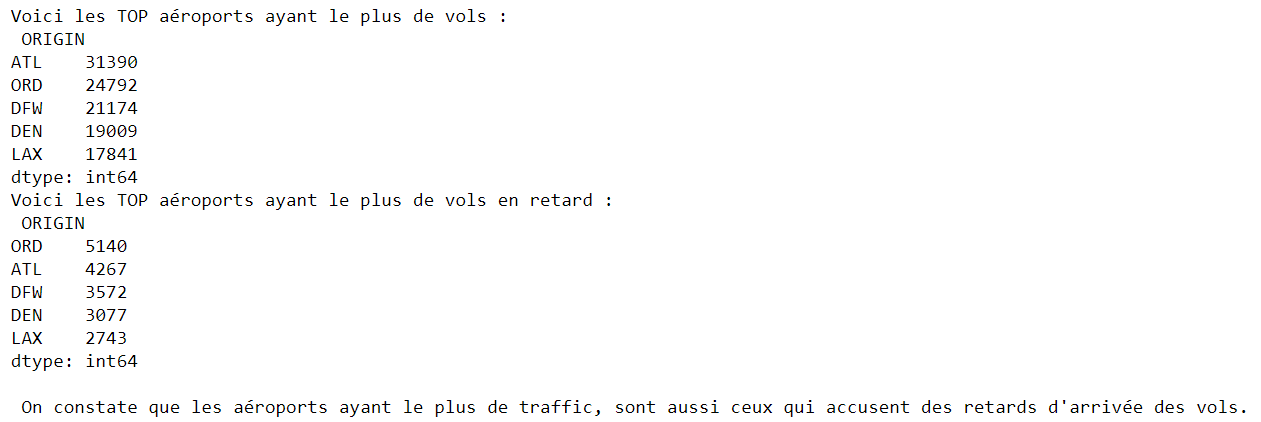
Constat : Les vols qui partent en fin de journée (18h) ou les vols qui arrivent en fin de journée (20h), sont très fortement en retard.

|  |
| --- |
| plt.hist(df\_delay.DISTANCE, range = (0, 4000), bins = 50, color = 'blue',  edgecolor = 'black')  plt.xlabel('Distance de vol')  plt.ylabel('nombre vols')  plt.title("Nbre de vols en retard selon la distance parcourue") |



Ce sont donc les petites distances qui comptent une grande partie des vols retardés.

|  |
| --- |
| # Voici les TOP aéroports ayant le plus de vols  nb\_vol\_origin = df\_vol.groupby("ORIGIN").size().nlargest(5)  print("Voici les TOP aéroports ayant le plus de vols : \n", nb\_vol\_origin)  # Voici les TOP aéroports ayant le plus de vols en retard  nb\_delay\_origin = df\_delay.groupby("ORIGIN").size().nlargest(5)  print("Voici les TOP aéroports ayant le plus de vols en retard : \n", nb\_delay\_origin)  print("\n On constate que les aéroports ayant le plus de traffic, sont aussi ceux qui accusent des retards d'arrivée des vols.") |



# Jointures





### **Matériels**

Comme précédemment dit, nous allons effectuer les opérations de jointure avec le langage SCALA. Pour débugger dans un premier temps, le plus simple est d’écrire les bouts de code sous un Jupyter Notebook avec le plugin SPYLON-KERNEL, avant de générer le fichier scala.

### **Programme scala**

* 1. Spark Session

La première étape consiste à créer une session Spark.

|  |
| --- |
| import org.apache.spark.sql.SparkSession  val spark = SparkSession  .builder  .appName("flight")  .config("spark.master", "local")  .getOrCreate()  import spark.implicits. |

* 1. Chargement des données

Puis nous créons une première classe pour les vols, une seconde classe pour la météo. Et nous chargeons les données nettoyées.

Pour les données de vol, nous ne garderons que les champs qui nous intéressent :

* L’aéroport d'origine,
* L’aéroport d'arrivée,
* Les horaires programmés et réels de décollage,
* Les horaires programmés et réels d'atterrissage,
* Et la constatation du retard > à 15 minutes.

Pour les données météo, nous garderons :

* t0 : l'horaire de l'observation,
* A : le lieu (qui doit correspondre dans notre code, à l'aéroport),
* Les valeurs des observations : H pour humidité, Wd pour la direction du vent, Ws pour la vitesse du vent, P pour la pression, T6 pour la température, S pour les conditions du ciel, D pour la description du phénomène météorologique, V pour la visibilité.

|  |
| --- |
| //val workingDir = "hdfs:/user/user160/flight/"  val workingDir = "/FileStore/tables/1M/”  case class flight(  ORIGIN : String,  DEST : String,  Tad:String,  Tsd:String,  Taa:String,  Tsa:String,  Th:Boolean  )  val FT = spark.read.format("csv")  .option("header","true")  .option("inferSchema","true")  .load(workingDir+"flight\_1M.csv")  .as[flight]  //FT.printSchema  //FT.show(2)  case class weather(  t0:String,  A: String,  H: Float,  Wd: Float,  Ws: Float,  P: Float,  T6: Float,  S: String,  D: String,  V: Float  )  val OT = spark.read.format("csv")  .option("header","true")  .option("inferSchema","true")  .option("timestampFormat", "yyyy-MM-dd'T'HH:mm:ss")  .load(workingDir+"weather\_1M.csv")  .withColumn("H",$"H".cast("float"))  .withColumn("Wd",$"Wd".cast("float"))  .withColumn("Ws",$"Ws".cast("float"))  .withColumn("P",$"P".cast("float"))  .withColumn("T6",$"T6".cast("float"))  .withColumn("V",$"V".cast("float"))  .as[weather]  //OT.printSchema  //OT.show(2) |

* 1. Convertir le format date

Face à la difficulté d’effectuer des opérations sur les horaires tels que fournis par les jeux de données, nous utilisons la fonction "**unix\_timestamp**" pour convertir en valeur numérique les horaires.

Par exemple, ici, on récupère pour chaque vol les valeurs numériques des horaires de décollage programmés et des horaires d'atterrissage programmés.

|  |
| --- |
| import spark.sqlContext.implicits.\_  import org.apache.spark.sql.types.\_  import spark.sqlContext.implicits.\_  // sur les données vols  //Convert timestamp to unix timestamp  val df = FT.select(  $"Tsa",  unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss").as("Tsa\_unix"),  from\_unixtime(unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-12\*3600).as("Tsa\_12")  )  df.printSchema()  df.show(false)  val data\_f = FT.select(  $"ORIGIN",  $"DEST",  unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss").as("Tsd\_unix"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-1\*3600).as("Tsd\_unix\_1"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-2\*3600).as("Tsd\_unix\_2"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-3\*3600).as("Tsd\_unix\_3"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-4\*3600).as("Tsd\_unix\_4"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-5\*3600).as("Tsd\_unix\_5"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-6\*3600).as("Tsd\_unix\_6"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-7\*3600).as("Tsd\_unix\_7"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-8\*3600).as("Tsd\_unix\_8"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-9\*3600).as("Tsd\_unix\_9"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-10\*3600).as("Tsd\_unix\_10"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-11\*3600).as("Tsd\_unix\_11"),  (unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-12\*3600).as("Tsd\_unix\_12"),  unix\_timestamp($"Tad","yyyy-MM-dd HH:mm:ss").as("Tad\_unix"),  unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss").as("Tsa\_unix"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-1\*3600).as("Tsa\_unix\_1"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-2\*3600).as("Tsa\_unix\_2"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-3\*3600).as("Tsa\_unix\_3"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-4\*3600).as("Tsa\_unix\_4"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-5\*3600).as("Tsa\_unix\_5"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-6\*3600).as("Tsa\_unix\_6"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-7\*3600).as("Tsa\_unix\_7"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-8\*3600).as("Tsa\_unix\_8"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-9\*3600).as("Tsa\_unix\_9"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-10\*3600).as("Tsa\_unix\_10"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-1213600).as("Tsa\_unix\_11"),  (unix\_timestamp($"Tsa","yyyy-MM-dd HH:mm:ss")-12\*3600).as("Tsa\_unix\_12"),  unix\_timestamp($"Taa","yyyy-MM-dd HH:mm:ss").as("Taa\_unix"),  $"Th")  // sur les données météo  val data\_w = OT.select(  $"A",  unix\_timestamp($"t0","yyyy-MM-dd HH:mm:ss").as("t0\_unix"),  $"t0",  $"H",  $"Wd",  $"Ws",  $"P",  $"T6",  $"S",  $"D",  $"V")  //data\_w.printSchema()  // sur les données météo,  // on met les observations météorologiques dans un tuple  val data\_weather = data\_w.select(  $"A",  $"t0",  $"t0\_unix",  array($"t0\_unix",$"H",$"Wd",$"Ws",$"P",$"T6",$"S",$"D",$"V").as("weatherInfo")  )  //data\_weather.printSchema()  //data\_weather.show(5) |

Nous faisons une petite vérification sur le calcul de l’horaire -12h. Nous portons une attention toute particulière sur le passage de la journée (minuit)  cela semble bien fonctionner.

|  |
| --- |
| val df = FT.select(  $"Tsd",  unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss").as("Tsd\_unix"),  from\_unixtime(unix\_timestamp($"Tsd","yyyy-MM-dd HH:mm:ss")-12\*3600).as("Tsd\_12")  )  //df.printSchema()  //df.show(false) |

* 1. Création des clefs de jointure

Afin de préparer la jointure entre les données de vols et les observations météorologiques, nous créons les clefs suivantes :

* Origine + horaire programmé de décollage originK
* Destination + horaire programmé d'atterrissage destK

|  |
| --- |
| val data\_flights = data\_f.select(  $"ORIGIN",  $"DEST",  concat($"ORIGIN",$"Tsd\_unix").as("originK"),  concat($"DEST",$"Tsa\_unix").as("destK"),  $"Tsd\_unix",  $"Tsd\_unix\_1",  $"Tsd\_unix\_2",  $"Tsd\_unix\_3",  $"Tsd\_unix\_4",  $"Tsd\_unix\_5",  $"Tsd\_unix\_6",  $"Tsd\_unix\_7",  $"Tsd\_unix\_8",  $"Tsd\_unix\_9",  $"Tsd\_unix\_10",  $"Tsd\_unix\_11",  $"Tsd\_unix\_12",  $"Tad\_unix",  $"Tsa\_unix",  $"Tsa\_unix\_1",  $"Tsa\_unix\_2",  $"Tsa\_unix\_3",  $"Tsa\_unix\_4",  $"Tsa\_unix\_5",  $"Tsa\_unix\_6",  $"Tsa\_unix\_7",  $"Tsa\_unix\_8",  $"Tsa\_unix\_9",  $"Tsa\_unix\_10",  $"Tsa\_unix\_11",  $"Tsa\_unix\_12",  $"Taa\_unix",  $"Th")  //data\_flights.show(5,false) |

* 1. Stratégie relative aux jointures et au découpage des créneaux horaires des observations météorologiques

Nous observons que la fréquence et le nombre de relevés météorologiques peuvent varier selons les stations météo. Pour une même tranche horaire, une station peut effectuer un seul relevé, tandis que d’autres (stations) peuvent en effectuer plusieurs, selons des fréquences qui leurs sont propres. Face à cette situation, il nous a fallut sélectioner pour chaque station et chaque tranche horaire de 60 minutes (relative à l’heure de départ ou d’arrivée d’un vol), une seule observation météorologique. Pour ce faire nous avons envisagé deux méthodes.

**Méthode 1 : une observation météo par aéroport et par tranche d’une heure**

Le principe est simple et permet d’avoir un seul set de données météo par heure et par station météo (on choisit arbitrairement une observation météo et une seule dans la plage de 60 minutes).

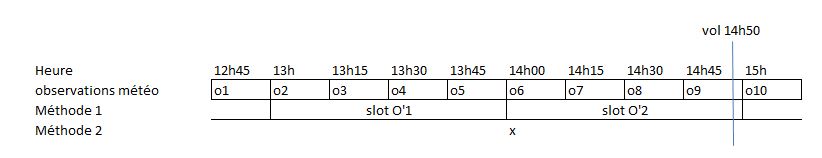
* **Avantages :** jointures directes sans besoin de rechercher l’horaire le plus proche entre heure\_vol et heure\_du\_vol-60 minutes. Donc une réduction très importante des « shuffle/sort » et de la quantitéde données météo à manipuler en mémoire puisqu’il y a une seule observation météo par heure.
* **Inconvénients :** on perd en précision horaire des observations météo, ce qui peut avoir un impact sur la précision du modèle lorsque la météo change brutalement en une heure.

**Méthode 2 : On regroupe l’ensemble des observations par plage horaire [heure\_vol-60min ; heure\_vol]** puis on les ordonnance par ordre chronologique avant de choisir celle la plus proche de la borne horaire inférieure.

* **Avantages :** l’observation météo pour chaque plage horaire est calibrée pour chaque heure de départ ou arrivée des vols. Cela devrait améliorer la précision des prévisions du modèle.
* **Inconvénients :** jointures complexes en passant par des listes d’observations par plage pour chaque heure de départ et d’arrivée des vols. Cela augmente très notablement les temps de calcul.

Voici un exemple pour illustrer :

Prenons le cas d’un vol à 14h50 et on détermine l’observation météo à prendre en compte dans la plage horaire [13h50 ; 14h50[



Méthode 1 : pour chaque plage horaire [(XX-1)h00;(XX)h00[, on choisit arbitrairement une seule observation météo représentative de la plage horaire, cela peut être par exemple l’observation moyenne ou en milieu de plage, dans cet exemple l’observation météo notée slot O’2

Méthode 2 : on choisit l’observation météo la plus proche de la borne inférieure de la plage horaire, dans notre cas cela sera l’observation météo notée“o6” (à 14h00)

L’arbitrage qui a été fait est de privilégier la précision des données en choisissant l’observation météo la plus proche de la borne inférieure de la plage temporelle, d’où le choix de la méthode 2. La contrepartie de ce choix est l’augmentation du temps de calcul. Toutefois, celui-ci reste dans la limite du raisonnable.

* 1. PREMIERE jointure : Aéroport et données météo

On distingue 2 cas :

**Aéroport de départ + données météorologiques** : On part de la table data\_flights, et on effectue une jointure pour récupérer les observations météo des aéroports d’origine, pour chacune des 12 tranches horaires  12 dataframes en sortie pour les aéroports d’origine.

**Aéroport d’arrivée + données météorologiques** : On refait de même pour les aéroports d’arrivé  on obtient 12 dataframes.

|  |
| --- |
| var Flights\_oA = data\_flights.join(data\_weather, (data\_flights("ORIGIN")===data\_weather("A")) && (data\_flights("Tsd\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsd\_unix\_1")<= data\_weather("t0\_unix")), "inner")  val Flights\_o1 =Flights\_oA.groupBy("originK").agg(sort\_array(collect\_list("weatherInfo"),asc = true).getItem(0).alias("slot1\_O")).withColumnRenamed("originK","originK1")  Flights\_oA = data\_flights.join(data\_weather, (data\_flights("ORIGIN")===data\_weather("A")) && (data\_flights("Tsd\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsd\_unix\_2")<= data\_weather("t0\_unix")) && (data\_flights("Tsd\_unix\_1")>= data\_weather("t0\_unix")) , "inner")  val Flights\_o2 =Flights\_oA.groupBy("originK").agg(sort\_array(collect\_list("weatherInfo"),true).getItem(0).alias("slot2\_O")).withColumnRenamed("originK","originK2")  Flights\_oA = data\_flights.join(data\_weather, (data\_flights("ORIGIN")===data\_weather("A")) && (data\_flights("Tsd\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsd\_unix\_3")<= data\_weather("t0\_unix")) &&(data\_flights("Tsd\_unix\_2")>= data\_weather("t0\_unix")) , "inner")  val Flights\_o3 =Flights\_oA.groupBy("originK").agg(sort\_array(collect\_list("weatherInfo"),true).getItem(0).alias("slot3\_O")).withColumnRenamed("originK","originK3")  …… |

|  |
| --- |
| var Flights\_dA = data\_flights.join(data\_weather, (data\_flights("DEST")===data\_weather("A")) && (data\_flights("Tsa\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsa\_unix\_1")<= data\_weather("t0\_unix")), "inner")  val Flights\_d1 =Flights\_dA.groupBy("destK").agg(sort\_array(collect\_list("weatherInfo"),asc = true).getItem(0).alias("slot1\_D")).withColumnRenamed("destK","destK1")  Flights\_dA = data\_flights.join(data\_weather, (data\_flights("DEST")===data\_weather("A")) && (data\_flights("Tsa\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsa\_unix\_2")<= data\_weather("t0\_unix")) && (data\_flights("Tsa\_unix\_1")>= data\_weather("t0\_unix")) , "inner")  val Flights\_d2 =Flights\_dA.groupBy("destK").agg(sort\_array(collect\_list("weatherInfo"),true).getItem(0).alias("slot2\_D")).withColumnRenamed("destK","destK2")  Flights\_dA = data\_flights.join(data\_weather, (data\_flights("DEST")===data\_weather("A")) && (data\_flights("Tsa\_unix")>= data\_weather("t0\_unix")) &&(data\_flights("Tsa\_unix\_3")<= data\_weather("t0\_unix")) &&(data\_flights("Tsa\_unix\_2")>= data\_weather("t0\_unix")) , "inner")  val Flights\_d3 =Flights\_dA.groupBy("destK").agg(sort\_array(collect\_list("weatherInfo"),true).getItem(0).alias("slot3\_D")).withColumnRenamed("destK","destK3")  …… |

* 1. SECONDE jointure : {Aéroport départ + météo} et {Aéroport arrivée + météo}

Pour chaque vol, on doit récupérer les données météorologiques des aéroports de départ et des aéroports d’arrivée, issues de la 1ère jointure.

|  |
| --- |
| val data\_flights\_part1 = data\_flights  // les 12 jointures finales pour l'aéroport d'origine  .join(Flights\_o1,data\_flights("originK")===Flights\_o1("originK1"),"left\_outer")  .join(Flights\_o2,data\_flights("originK")===Flights\_o2("originK2"),"left\_outer")  .join(Flights\_o3,data\_flights("originK")===Flights\_o3("originK3"),"left\_outer")  .join(Flights\_o4,data\_flights("originK")===Flights\_o4("originK4"),"left\_outer")  .join(Flights\_o5,data\_flights("originK")===Flights\_o5("originK5"),"left\_outer")  .join(Flights\_o6,data\_flights("originK")===Flights\_o6("originK6"),"left\_outer")  .join(Flights\_o7,data\_flights("originK")===Flights\_o7("originK7"),"left\_outer")  .join(Flights\_o8,data\_flights("originK")===Flights\_o8("originK8"),"left\_outer")  .join(Flights\_o9,data\_flights("originK")===Flights\_o9("originK9"),"left\_outer")  .join(Flights\_o10,data\_flights("originK")===Flights\_o10("originK10"),"left\_outer")  .join(Flights\_o11,data\_flights("originK")===Flights\_o11("originK11"),"left\_outer")  .join(Flights\_o12,data\_flights("originK")===Flights\_o12("originK12"),"left\_outer")  // Les 12 jointures finales pour l'aéroport destination  .join(Flights\_d1,data\_flights("destK")===Flights\_d1("destK1"),"left\_outer")  .join(Flights\_d2,data\_flights("destK")===Flights\_d2("destK2"),"left\_outer")  .join(Flights\_d3,data\_flights("destK")===Flights\_d3("destK3"),"left\_outer")  .join(Flights\_d4,data\_flights("destK")===Flights\_d4("destK4"),"left\_outer")  .join(Flights\_d5,data\_flights("destK")===Flights\_d5("destK5"),"left\_outer")  .join(Flights\_d6,data\_flights("destK")===Flights\_d6("destK6"),"left\_outer")  .join(Flights\_d7,data\_flights("destK")===Flights\_d7("destK7"),"left\_outer")  .join(Flights\_d8,data\_flights("destK")===Flights\_d8("destK8"),"left\_outer")  .join(Flights\_d9,data\_flights("destK")===Flights\_d9("destK9"),"left\_outer")  .join(Flights\_d10,data\_flights("destK")===Flights\_d10("destK10"),"left\_outer")  .join(Flights\_d11,data\_flights("destK")===Flights\_d11("destK11"),"left\_outer")  .join(Flights\_d12,data\_flights("destK")===Flights\_d12("destK12"),"left\_outer")  // suppression des champs intermédiaires désormais inutiles  .drop("destK1","destK2","destK3","destK4","destK5","destK6","destK7","destK8","destK9","destK10","destK11","destK12")  val test = data\_flights\_part1.cache()  test.printSchema() |

* 1. Sauvegarde des résultats en PARQUET

Nous nous sommes demandé sous quel format nous allions sauvegarder le dataframe issue des jointures. Dans un premier temps, nous avons sauvegardé nos données au format json, puis nous avons finalement opté pour un enregistrement des résultats au format **PARQUET**, particulièrement adapté aux données massives.

Au format parquet, les données sont présentées en colonne, contrairement aux formats csv ou json où les données sont présentées lignes par lignes.

Autrement dit, toutes les données du premier attribut sont stockées, puis seulement les données du second attribut, etc.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DEN | CLR | AAA | … | 1325683800 | 1325652600 | 1325690700 |  |  |  |  |  |  |  |  |  |  |  |
| L1 | L2 |  |  | L1 | L2 |  |  | L1 | L2 |  |  | L1 | L2 |  |  |  |  |

Les avantages sont multiples :

Représentation des données en colonnes

* Lors de la lecture ou de l’écriture d’un attribut les opérations sont facilitées car les données se trouvent regroupées.
* Pour atteindre des attributs précis, il suffit de « sauter » les attributs que l’on ne souhaite pas.
* La représentation des valeurs manquantes (ou nulles) a un coût nettement plus faible qu’avec une représentation en ligne.

Division en pages et en blocs

* Un jeu de données est divisé en pages. Chaque page est elle-même constituée de blocs contenant les colonnes d’un certain nombre d’entrées. Les blocs sont organisés en colonnes comme indiqué précédemment. Une page a généralement une taille comprise entre 100 Mo et 1 Go, de sorte que les fichiers soient aisément manipulables. Cela facilite également le partage des données, la communication d’une page d’une unité de traitement à une autre ne prenant que quelques secondes sur un réseau informatique moderne.
* À chaque page (et chaque bloc) sont associées des métadonnées. Il s’agit de statistiques décrivant le contenu de la page (ou du bloc), telles que les valeurs minimales et maximales de chaque colonne. Ces métadonnées sont utilisées afin d’accélérer les recherches réalisées lors d’une analyse.

Encodage pour alléger les données

* Format binaire adapté pour le CPU
* L’encodage est spécifique à chaque page. Par exemple, si une colonne possède peu de valeurs distinctes, alors Parquet choisit un encodage en dictionnaire. De même, encoder une variation de valeurs plutôt que les valeurs elle même peut s’avérer plus efficace. Autre exemple, pour les textes, on peut choisir d’encoder uniquement les suffixes qui diffèrent….

Compression

* Compression des données avec Snappy qui est optimisé pour la vitesse d’exécution.

Au format PARQUET

|  |
| --- |
| test.coalesce(4).write.parquet(workingDir+"flight\_1M.parquet")  val JT\_parquet = spark.read.parquet(workingDir + "flight\_1M.parquet")  JT\_parquet.printSchema()  display(JT\_parquet)  dbutils.fs.rm("/FileStore/tables/1M/weather\_1M.csv")  dbutils.fs.rm("/FileStore/tables/1M/flight\_1M.csv") |

A ce stade, les données ont les types suivants :

|  |  |
| --- | --- |
| STRING | ORIGIN, DEST, originK, destK, Th |
| LONG | Tsd-unix, Tsd\_unix\_1, Tsd\_unix\_2, …  Tsa-unix, Tsa\_unix\_1, Tsa\_unix\_2, …  Tad\_unix, Taa\_unix  Tsa\_unix |
| ARRAY(STRING) | slot1\_O, slot2\_O, slot3\_O, …  slot1\_O, slot2\_O, slot3\_O, … |

# Pipeline et modèles







## Implémentation du pipeline et du modèle à entrainer

Le pipeline ainsi que la partie de prédiction se trouvent dans un notebook à part, toujours avec le plug in « Spylon-kernel » sur JupyterNotebook, ou directement en scala sous Databricks.

### **Préparation des données avant Pipeline**

Comme d’habitude, il faut lancer une session Spark :

|  |
| --- |
| import org.apache.spark.sql.SparkSession  val spark = SparkSession  .builder  .appName("flight")  .config("spark.master", "local")  .getOrCreate()  import spark.implicits.\_ |

Ensuite on charge le fichier au format parquet.

*Nous avons dans un premier temps effectué des tests de fonctionnement de notre code sur une petite quantité de données (seulement 100 lignes), le but étant de vérifier son fonctionnement. Le jeu de données entier sera testé en version distribuée après.*

|  |
| --- |
| val workingDir ="/FileStore/tables/6M/"  val JT\_parquet = spark.read.parquet(workingDir + "flight\_6M.parquet")  .withColumn("Th",$"Th".cast("string"))  val JT = JT\_parquet//.limit(100) |

Nous allons maintenant équilibrer les données  Cette opération nous permettra de réduire la quantité de données qui sera traitée par la suite.

L’équilibrage est nécessaire. En effet, il y a davantage de vols à l’heure, qu’en retard (heureusement !). Pour aboutir à notre équilibrage, nous comptons les vols en retard, et nous en sélectionnant le même nombre, au hasard, de vols à l’heure. Nous avons ainsi des données équilibrées.

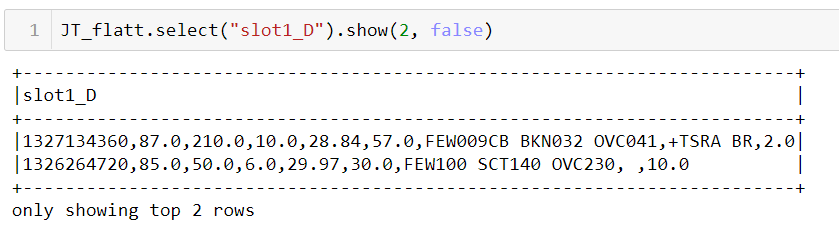
|  |
| --- |
| JT.printSchema()  // on compte le nbre total de lignes dans nos données  val nb\_total = JT.count  // on compte le nbre de label en retard (ie 1.0)  val nb\_delay = JT.filter(JT ("Th") === "true").count.toInt  // on compte le nbre de label à l'heure (ie 0.0)  val nb\_onTime = JT.filter(JT ("Th") === "false").count.toInt  // on prend le plus petit nbre  val nb\_selection = Math.min(nb\_onTime, nb\_delay)  // on mélange nos données  import org.apache.spark.sql.functions.rand  val JT\_ok\_shuffled = JT.orderBy(rand())  // on prend autant en retard que on time  val JT\_ok\_shuffled\_delay=JT\_ok\_shuffled.filter(JT\_ok\_shuffled("Th") === "true").limit(nb\_selection)  val JT\_ok\_shuffled\_onTime=JT\_ok\_shuffled.filter(JT\_ok\_shuffled("Th") === "false").limit(nb\_selection)  val JT\_balanced = JT\_ok\_shuffled\_delay.unionAll(JT\_ok\_shuffled\_onTime)  //JT\_balanced.show(5) |

Nous modifions nos champs SLOTs qui sont des array(strings) vers des strings séparés par des virgules.

On en profite pour réduire le nombre d’attributs pour sélectionner uniquement ceux qui nous seront utiles.

|  |
| --- |
| val convertToString = udf((array: Seq[String]) => {array.mkString(",")})  val JT\_flatt = JT\_balanced.na.drop()  .withColumn("slot1\_D", convertToString($"slot1\_D"))  .withColumn("slot2\_D", convertToString($"slot2\_D"))  .withColumn("slot3\_D", convertToString($"slot3\_D"))  .withColumn("slot4\_D", convertToString($"slot4\_D"))  .withColumn("slot5\_D", convertToString($"slot5\_D"))  .withColumn("slot6\_D", convertToString($"slot6\_D"))  .withColumn("slot7\_D", convertToString($"slot7\_D"))  .withColumn("slot8\_D", convertToString($"slot8\_D"))  .withColumn("slot9\_D", convertToString($"slot9\_D"))  .withColumn("slot10\_D", convertToString($"slot10\_D"))  .withColumn("slot11\_D", convertToString($"slot11\_D"))  .withColumn("slot12\_D", convertToString($"slot12\_D"))  .withColumn("slot1\_A", convertToString($"slot1\_A"))  .withColumn("slot2\_A", convertToString($"slot2\_A"))  .withColumn("slot3\_A", convertToString($"slot3\_A"))  .withColumn("slot4\_A", convertToString($"slot4\_A"))  .withColumn("slot5\_A", convertToString($"slot5\_A"))  .withColumn("slot6\_A", convertToString($"slot6\_A"))  .withColumn("slot7\_A", convertToString($"slot7\_A"))  .withColumn("slot8\_A", convertToString($"slot8\_A"))  .withColumn("slot9\_A", convertToString($"slot9\_A"))  .withColumn("slot10\_A", convertToString($"slot10\_A"))  .withColumn("slot11\_A", convertToString($"slot11\_A"))  .withColumn("slot12\_A", convertToString($"slot12\_A"))  .select("Th", "ORIGIN", "DEST",  "slot1\_D", "slot2\_D", "slot3\_D", "slot4\_D", "slot5\_D", "slot6\_D", "slot7\_D", "slot8\_D", "slot9\_D", "slot10\_D", "slot11\_D", "slot12\_D",  "slot1\_A", "slot2\_A", "slot3\_A", "slot4\_A", "slot5\_A", "slot6\_A", "slot7\_A", "slot8\_A", "slot9\_A", "slot10\_A", "slot11\_A", "slot12\_A") |

A cette étape, notre jeu de données n’est pas encore complétement prêt. Les différents slots sont au format ci-dessous.

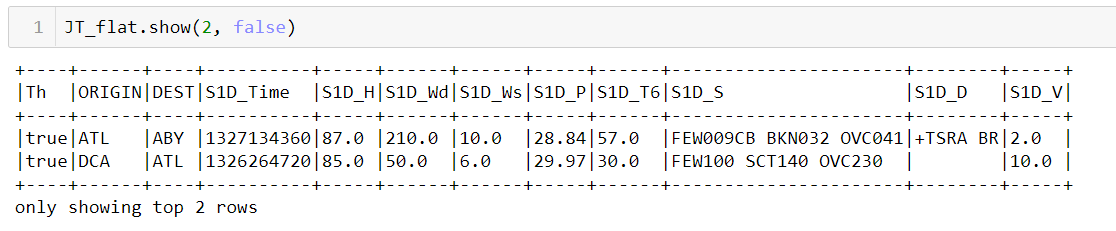


Il nous faut maintenant séparer les éléments de chaque slot en utilisant la virgule comme séparateur, et mettre chaque attribut dans une colonne spécifique.

En voici le code :

|  |
| --- |
| val JT\_flat = JT\_flatt  // niveau 1 du getItem  .withColumn("S1D\_Time",split(col("slot1\_D"), ",").getItem(0))  .withColumn("S1D\_H",split(col("slot1\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S1D\_Wd",split(col("slot1\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S1D\_Ws",split(col("slot1\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S1D\_P",split(col("slot1\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S1D\_T6",split(col("slot1\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S1D\_S",split(col("slot1\_D"), ",").getItem(6))  .withColumn("S1D\_D",split(col("slot1\_D"), ",").getItem(7))  .withColumn("S1D\_V",split(col("slot1\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S2D\_Time",split(col("slot2\_D"), ",").getItem(0))  .withColumn("S2D\_H",split(col("slot2\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S2D\_Wd",split(col("slot2\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S2D\_Ws",split(col("slot2\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S2D\_P",split(col("slot2\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S2D\_T6",split(col("slot2\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S2D\_S",split(col("slot2\_D"), ",").getItem(6))  .withColumn("S2D\_D",split(col("slot2\_D"), ",").getItem(7))  .withColumn("S2D\_V",split(col("slot2\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S3D\_Time",split(col("slot3\_D"), ",").getItem(0))  .withColumn("S3D\_H",split(col("slot3\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S3D\_Wd",split(col("slot3\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S3D\_Ws",split(col("slot3\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S3D\_P",split(col("slot3\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S3D\_T6",split(col("slot3\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S3D\_S",split(col("slot3\_D"), ",").getItem(6))  .withColumn("S3D\_D",split(col("slot3\_D"), ",").getItem(7))  .withColumn("S3D\_V",split(col("slot3\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S4D\_Time",split(col("slot4\_D"), ",").getItem(0))  .withColumn("S4D\_H",split(col("slot4\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S4D\_Wd",split(col("slot4\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S4D\_Ws",split(col("slot4\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S4D\_P",split(col("slot4\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S4D\_T6",split(col("slot4\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S4D\_S",split(col("slot4\_D"), ",").getItem(6))  .withColumn("S4D\_D",split(col("slot4\_D"), ",").getItem(7))  .withColumn("S4D\_V",split(col("slot4\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S5D\_Time",split(col("slot5\_D"), ",").getItem(0))  .withColumn("S5D\_H",split(col("slot5\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S5D\_Wd",split(col("slot5\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S5D\_Ws",split(col("slot5\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S5D\_P",split(col("slot5\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S5D\_T6",split(col("slot5\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S5D\_S",split(col("slot5\_D"), ",").getItem(6))  .withColumn("S5D\_D",split(col("slot5\_D"), ",").getItem(7))  .withColumn("S5D\_V",split(col("slot5\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S6D\_Time", split(col("slot6\_D"), ",").getItem(0))  .withColumn("S6D\_H", split(col("slot6\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S6D\_Wd", split(col("slot6\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S6D\_Ws", split(col("slot6\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S6D\_P", split(col("slot6\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S6D\_T6", split(col("slot6\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S6D\_S", split(col("slot6\_D"), ",").getItem(6))  .withColumn("S6D\_D", split(col("slot6\_D"), ",").getItem(7))  .withColumn("S6D\_V", split(col("slot6\_D"), ",").getItem(8).cast(DoubleType))  .withColumn("S7D\_Time", split(col("slot7\_D"), ",").getItem(0))  .withColumn("S7D\_H", split(col("slot7\_D"), ",").getItem(1).cast(DoubleType))  .withColumn("S7D\_Wd", split(col("slot7\_D"), ",").getItem(2).cast(DoubleType))  .withColumn("S7D\_Ws", split(col("slot7\_D"), ",").getItem(3).cast(DoubleType))  .withColumn("S7D\_P", split(col("slot7\_D"), ",").getItem(4).cast(DoubleType))  .withColumn("S7D\_T6", split(col("slot7\_D"), ",").getItem(5).cast(DoubleType))  .withColumn("S7D\_S", split(col("slot7\_D"), ",").getItem(6))  .withColumn("S7D\_D", split(col("slot7\_D"), ",").getItem(7))  .withColumn("S7D\_V", split(col("slot7\_D"), ",").getItem(8).cast(DoubleType))  .drop("slot1\_D","slot2\_D", "slot3\_D", "slot4\_D" , "slot5\_D", "slot6\_D", "slot7\_D", "slot8\_D", "slot9\_D", "slot10\_D", "slot11\_D", "slot12\_D")  .withColumn("S1A\_Time", split(col("slot1\_A"), ",").getItem(0))  .withColumn("S1A\_H", split(col("slot1\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S1A\_Wd", split(col("slot1\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S1A\_Ws", split(col("slot1\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S1A\_P", split(col("slot1\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S1A\_T6", split(col("slot1\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S1A\_S", split(col("slot1\_A"), ",").getItem(6))  .withColumn("S1A\_D", split(col("slot1\_A"), ",").getItem(7))  .withColumn("S1A\_V", split(col("slot1\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S2A\_Time", split(col("slot2\_A"), ",").getItem(0))  .withColumn("S2A\_H", split(col("slot2\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S2A\_Wd", split(col("slot2\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S2A\_Ws", split(col("slot2\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S2A\_P", split(col("slot2\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S2A\_T6", split(col("slot2\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S2A\_S", split(col("slot2\_A"), ",").getItem(6))  .withColumn("S2A\_D", split(col("slot2\_A"), ",").getItem(7))  .withColumn("S2A\_V", split(col("slot2\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S3A\_Time", split(col("slot3\_A"), ",").getItem(0))  .withColumn("S3A\_H", split(col("slot3\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S3A\_Wd", split(col("slot3\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S3A\_Ws", split(col("slot3\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S3A\_P", split(col("slot3\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S3A\_T6", split(col("slot3\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S3A\_S", split(col("slot3\_A"), ",").getItem(6))  .withColumn("S3A\_D", split(col("slot3\_A"), ",").getItem(7))  .withColumn("S3A\_V", split(col("slot3\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S4A\_Time", split(col("slot4\_A"), ",").getItem(0))  .withColumn("S4A\_H", split(col("slot4\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S4A\_Wd", split(col("slot4\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S4A\_Ws", split(col("slot4\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S4A\_P", split(col("slot4\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S4A\_T6", split(col("slot4\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S4A\_S", split(col("slot4\_A"), ",").getItem(6))  .withColumn("S4A\_D", split(col("slot4\_A"), ",").getItem(7))  .withColumn("S4A\_V", split(col("slot4\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S5A\_Time", split(col("slot5\_A"), ",").getItem(0))  .withColumn("S5A\_H", split(col("slot5\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S5A\_Wd", split(col("slot5\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S5A\_Ws", split(col("slot5\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S5A\_P", split(col("slot5\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S5A\_T6", split(col("slot5\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S5A\_S", split(col("slot5\_A"), ",").getItem(6))  .withColumn("S5A\_D", split(col("slot5\_A"), ",").getItem(7))  .withColumn("S5A\_V", split(col("slot5\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S6A\_Time", split(col("slot6\_A"), ",").getItem(0))  .withColumn("S6A\_H", split(col("slot6\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S6A\_Wd", split(col("slot6\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S6A\_Ws", split(col("slot6\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S6A\_P", split(col("slot6\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S6A\_T6", split(col("slot6\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S6A\_S", split(col("slot6\_A"), ",").getItem(6))  .withColumn("S6A\_D", split(col("slot6\_A"), ",").getItem(7))  .withColumn("S6A\_V", split(col("slot6\_A"), ",").getItem(8).cast(DoubleType))  .withColumn("S7A\_Time", split(col("slot7\_A"), ",").getItem(0))  .withColumn("S7A\_H", split(col("slot7\_A"), ",").getItem(1).cast(DoubleType))  .withColumn("S7A\_Wd", split(col("slot7\_A"), ",").getItem(2).cast(DoubleType))  .withColumn("S7A\_Ws", split(col("slot7\_A"), ",").getItem(3).cast(DoubleType))  .withColumn("S7A\_P", split(col("slot7\_A"), ",").getItem(4).cast(DoubleType))  .withColumn("S7A\_T6", split(col("slot7\_A"), ",").getItem(5).cast(DoubleType))  .withColumn("S7A\_S", split(col("slot7\_A"), ",").getItem(6))  .withColumn("S7A\_D", split(col("slot7\_A"), ",").getItem(7))  .withColumn("S7A\_V", split(col("slot7\_A"), ",").getItem(8).cast(DoubleType))  .drop("slot1\_A", "slot2\_A", "slot3\_A","slot4\_A" , "slot5\_A", "slot6\_A", "slot7\_A", "slot8\_A", "slot9\_A", "slot10\_A", "slot11\_A", "slot12\_A") |

A ce stade, nous obtenons le jeu de données ci-dessous.



### **Pipeline**

Afin de bâtir notre pipeline, nous commençons par charger les librairies utiles.

|  |
| --- |
| import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel, Tokenizer}  import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer, VectorAssembler} //, OneHotEncoderEstimator}  import org.apache.spark.ml.feature.OneHotEncoder // hbc le 9 oct 2020 : OneHotEncoderEstimator est une version trop récente  import org.apache.spark.ml.Pipeline  import org.apache.spark.ml.PipelineModel  import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier, LogisticRegression, LogisticRegressionModel, GBTClassificationModel, GBTClassifier}  import org.apache.spark.ml.evaluation.{MulticlassClassificationEvaluator, BinaryClassificationEvaluator}  import org.apache.spark.ml.tuning.{CrossValidator, TrainValidationSplit, ParamGridBuilder,CrossValidatorModel}  import org.apache.spark.ml.param.ParamMap  import org.apache.spark.sql.types.DoubleType  import org.apache.spark.sql.functions.\_ |

a. Attributs SkyConditions et Descriptions

Pour les attributs « SkyConditions » et « Descriptions » qui correspondent respectivement aux colonnes « S1D\_D » et « S1D\_V », nous allons dans un premier temps explorer la structure particulière de ces données.

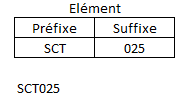
Ces champs de textes sont la concaténation de plusieurs éléments séparés par un espace.

a.1 Colonne Skyconditions (S)

Chaque élément est composé d’un descripteur et d’une distance, structuré de la manière suivante :

* Un préfixe de longueur variable : un descripteur météo en lettres codé sur 3 ou 2 caractères. Sauf pour la valeur ‘CLR’
* Un suffixe de longueur fixe : une distance en chiffre sur 3 caractères.

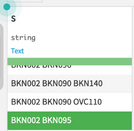
En voici un exemple :



En ce qui concerne le préfixe, voici le tableau de correspondance :



Ci-dessus une illustration du contenu du champ Skyconditions (S) :



On pourra remarquer les points singuliers suivants :

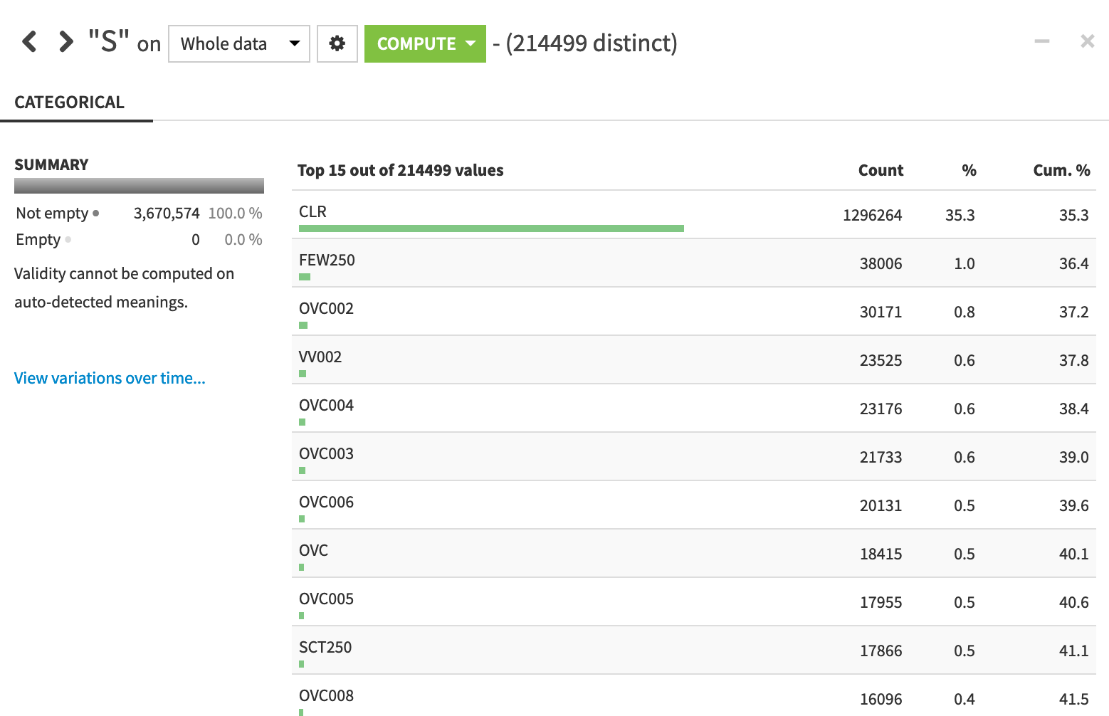
* Les éléments sont classés par ordre alphabétique.
* Un même descripteur (BKN) apparaît plusieurs fois avec des distances différentes.
* Ce champ a une structure de données complexe et variable, qui rend difficile une extraction des éléments les plus intéressants.

Exploration des données Skyconditions (S) :

Pour cette étape nous avons utilisé la plateforme IA Data Iku 

Sur les deux années 2012-2013 :

* Plus de 316.000 combinaisons différentes d’éléments
* 100% de lignes renseignées (pas de ligne vide)
* 35% de valeur ‘CLR’ seule
* Les autres combinaisons d’éléments sont très nombreuses mais très peu fréquentes
* Les champs ne comportent pas d’erreur de typographie.

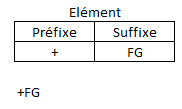


a.2 Colonne Descriptions (D)

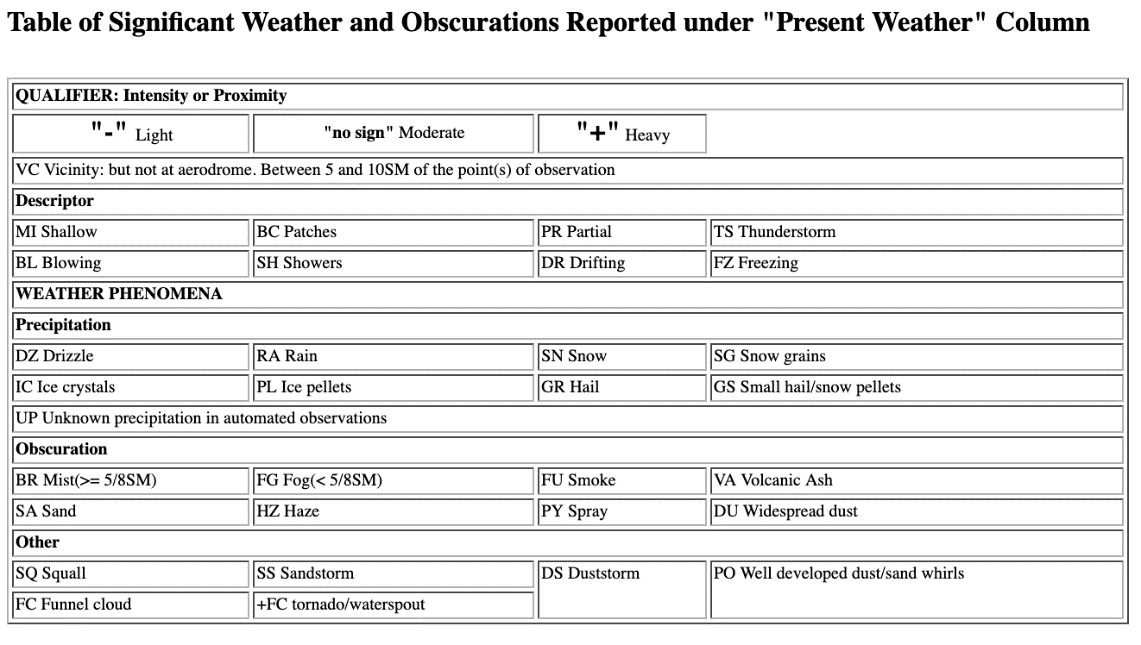
Selon la documentation officielle, en théorie chaque élément est composé d’un indice et d’un descripteur, structuré de la manière suivante :

* Un préfixe de longueur variable : un indicateur d’intensité météorologique, en lettre codé sur 0 ou 1 caractère (cf. Tableau plus bas)
* Un suffixe de longueur fixe : un descripteur en chiffre sur 2 caractères.

Un exemple :



En ce qui concerne les éléments, voici le tableau de correspondance :



Ci-dessous une illustration du contenu du champ Descriptions (D)



On pourra remarquer les points singuliers suivants :

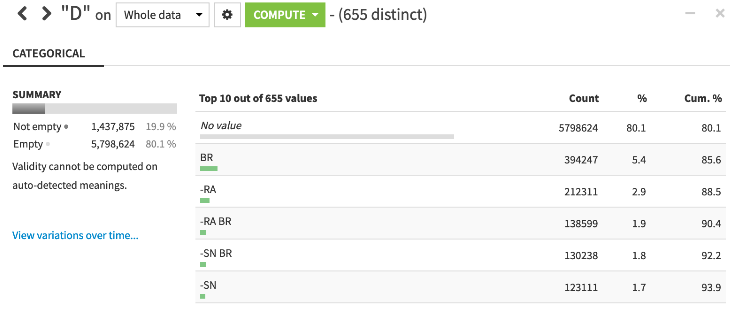
* Les champs comportent des erreurs de typographie (en rouge), le séparateur n'est parfois pas présent. La structure des données n’est pas très homogène (saisie manuelle ?).
* Ce champ a une structure de données complexe et variable, qui rend difficile une extraction des éléments les plus intéressants.

Exploration des données Descriptions (D) :

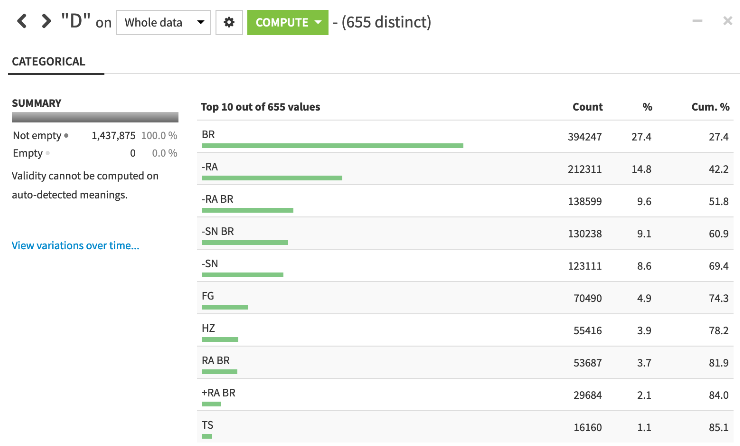
Pour cette étape nous avons utilisé la plateforme IA Data Iku 

Sur les deux années 2012-2013 :

* 655 combinaisons différentes d’éléments
* Données éparses, 80% de lignes vides



Voici l’histogramme (top 10) des champs non vides :



a.3 Méthode 1 : **Tokenizer()** et **CountVectorizer().**

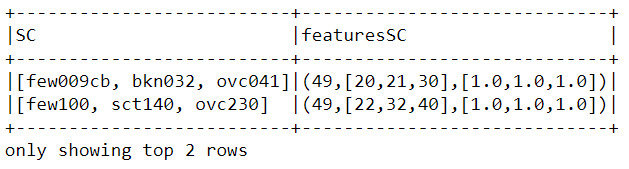
Nous appliquerons les fonctions Spark**Tokenizer()** et **CountVectorizer()**

|  |
| --- |
| val tokenizerSC = new Tokenizer()  .setInputCol("S1D\_S")  .setOutputCol("SC")  val tokenizerD = new Tokenizer()  .setInputCol("S1D\_D")  .setOutputCol("Description")  val countVectorizerSC = new CountVectorizer()  .setInputCol("SC")  .setOutputCol("featuresSC")  val countVectorizerD = new CountVectorizer()  .setInputCol("Description")  .setOutputCol("featuresD") |

A cette étape là, dans notre jeu de données, les attributs « SkyCondition » et « Descriptions » ont subi des transformations.

Par exemple, voici ce que devient « SkyCondition »  le tokenizer donne le colonne « SC », et le CountVectorizer donne la colonne « featuresSC » :

* + - Nous avons un vocabulaire de 49 mots, autrement dit, il y a 49 classes différentes dans l’attribut « SkyConditions ».
    - Nous avons également l’information que la valeur few009cb correspond à l’index 20, que bkn032 correspond à l’index 21, que ovc041 correspond à l’index 30.
    - Nous avons enfin la fréquence de ces 3 attributs dans la ligne concernée  on trouve des « 1 » ce qui est tout à fait logique.



a.3 Méthode 2 : Encodage simplifié

Voici l’exemple de l’encodage simplifié de la colonne Skyconditions (S)

Comme nous l’avons vu précédemment, ce champ a une structure complexe. On peut donc essayer de simplifier son interprétation de la manière suivante :

* Ne pas tenir compte du suffixe (distance codée sur 3 caractères)
* Se focaliser sur quelques descripteurs météo ayant potentiellement le plus d’impact sur les retards.

Différents tests ont été menés pour essayer d’identifier les valeurs principales à indexer permettant d’augmenter la précision du modèle.

C’est un encodage très simple en 3 valeurs :

* Ciel clair (‘CLR’ ou ‘SK’) => 2
* Visibilité verticale limitée => 0
* Autres cas => 1

// Encodeur du champ S (Sky condition)

val coder = udf((myVar:String) => {

if (myVar.contains("VV")) 0.0

if (myVar.contains("SK") || myVar.contains("CLR")) 2.0

else 1.0

})

a.4 Méthode 3 : Suppression des champs S & D

Dans notre étude expérimentale, nous allons ensuite comparer l’impact de ces trois approches différentes sur la performance du modèle.

b. Les autres attributs

Nous appliquons la fonction StringIndexer() :

|  |
| --- |
| ///////////// StringIndexer()  val labName = "Th"  val labelIndexer = new StringIndexer()  .setInputCol(labName)  .setOutputCol("indexed\_" + labName)  .setHandleInvalid("keep") // ajout pour éviter l'erreur au moment du calcul d'accuracy  val featuresCols = Array("S1D\_H","S1D\_Wd","S1D\_Ws","S1D\_P","S1D\_T6","S1D\_V",  "S2D\_H", "S2D\_Wd","S2D\_Ws","S2D\_P","S2D\_T6","S2D\_V",  "S3D\_H", "S3D\_Wd","S3D\_Ws","S3D\_P","S3D\_T6","S3D\_V",  "S4D\_H", "S4D\_Wd","S4D\_Ws","S4D\_P","S4D\_T6","S4D\_V",  "S5D\_H", "S5D\_Wd","S5D\_Ws","S5D\_P","S5D\_T6","S5D\_V",  "S6D\_H", "S6D\_Wd","S6D\_Ws","S6D\_P","S6D\_T6","S6D\_V",  "S7D\_H", "S7D\_Wd","S7D\_Ws","S7D\_P","S7D\_T6","S7D\_V",  "S1A\_H", "S1A\_Wd","S1A\_Ws","S1A\_P","S1A\_T6","S1A\_V",  "S2A\_H", "S2A\_Wd","S2A\_Ws","S2A\_P","S2A\_T6","S2A\_V",  "S3A\_H", "S3A\_Wd","S3A\_Ws","S3A\_P","S3A\_T6","S3A\_V",  "S4A\_H", "S4A\_Wd","S4A\_Ws","S4A\_P","S4A\_T6","S4A\_V",  "S5A\_H", "S5A\_Wd","S5A\_Ws","S5A\_P","S5A\_T6","S5A\_V",  "S6A\_H", "S6A\_Wd","S6A\_Ws","S6A\_P","S6A\_T6","S6A\_V",  "S7A\_H", "S7A\_Wd","S7A\_Ws","S7A\_P","S7A\_T6","S7A\_V") // sans S & D |

c. Maintenant, tous les attributs ensemble

|  |
| --- |
| val featureIndexer = new VectorAssembler()  .setInputCols(featuresCols)  .setOutputCol("indexed\_Features")  .setHandleInvalid("skip" |

d. Modèle testé : Random Forest Classifier

Avant de pouvoir lancer un quelconque algorithme d’apprentissage sur notre jeu de données, on sépare le jeu en training set et en test set.

|  |
| --- |
| val Array(trainingData, testData) = JT\_flat.randomSplit(Array(0.7, 0.3), seed = 12345)  trainingData.printSchema() |

Voici la création du random forest :

|  |
| --- |
| val rf = new RandomForestClassifier()  .setLabelCol("indexed\_Th")  .setFeaturesCol("featuresVecIndexed") |

Ainsi, notre pipeline COMPLET est le suivant :

|  |
| --- |
| //val labelConverter = new IndexToString()  // .setInputCol("prediction")  // .setOutputCol("predictedLabel")  // .setLabels(labelIndexer.labels)  // COMMAND ----------  val steps = Array(labelIndexer, featureIndexer, rf)  val pipeline = new Pipeline().setStages(steps) |

### **Les hyperparamètres et l’évaluation du modèle**

Nous allons maintenant faire intervenir un outil de **ParamGridBuilder().** Cette classe, combinée avec une fonction d’évaluation, ainsi qu’un **CrossValidator(),** vont nous permettre de tester plusieurs hyper paramètres des éléments utilisés précédemment, et d’en déterminer le meilleur.

**Remarque :** Nous aurions également pu utiliser la fonction **TrainValidationSplit()** en lieu et place de **Crossvalidator().** Cependant, la documentation proposée par Spark nous indique que TrainValidationSplit n’évalue chaque combinaison de paramètres qu’une seule fois, par opposition à k fois dans le cas de CrossValidator.

<https://spark.apache.org/docs/latest/ml-tuning.html>

Dans un premier temps, définissons les différentes valeurs des hyperparamètres à tester. Pour connaitre les hyper paramètres des différents éléments du pipeline, il suffit d’aller dans la documentation de Spark sur ces éléments, et rechercher les paramètres internes (IntParam).

Voici les hyper paramètres que nous proposons :

|  |
| --- |
| val paramGrid\_rf = new ParamGridBuilder()  .addGrid(rf.maxBins, Array(200))  .addGrid(rf.maxDepth, Array(25))  .addGrid(rf.numTrees, Array(150))  .addGrid(rf.impurity, Array("entropy"))// "gini" ou "entropy"  .build() |

Voici la fonction d’évaluation que nous allons utiliser :

|  |
| --- |
| val evaluator = new MulticlassClassificationEvaluator()  .setLabelCol("indexed\_Th")  .setPredictionCol("prediction")  .setMetricName("accuracy") // default = f1 |

Voici la fonction de CrossValidator() avec 3 folders :

|  |
| --- |
| val crossval = new CrossValidator()  .setEstimator(pipeline)  .setEvaluator(evaluator)  .setEstimatorParamMaps(paramGrid\_rf)  .setNumFolds(3) |

Nous « fittons » ce premier modèle « Random Forest Classifier » sur nos données de training :

|  |
| --- |
| val modelRF\_Grid = crossval.fit(trainingData) |

Quels sont les meilleurs valeurs d’hyperparamètres ?

|  |
| --- |
| modelRF\_Grid.getEstimatorParamMaps.zip(modelRF\_Grid.avgMetrics).maxBy(\_.\_2).\_1 |

Pour notre modèle de Random Forest Classifier, sur le petit jeu de données réduits, nous obtenons les valeurs suivantes :

{

rfc\_04bf9843fc88-impurity: entropy,

rfc\_04bf9843fc88-maxBins: 100,

vecIdx\_b1d1f033eddb-maxCategories: 2,

rfc\_04bf9843fc88-maxDepth: 10,

rfc\_04bf9843fc88-numTrees: 5,

cntVec\_78816e18599a-vocabSize: 100,

cntVec\_e4922ffb9d8c-vocabSize: 100

}

Nous évaluons maintenant la performance de ce modèle, sur les données de test :

|  |
| --- |
| val predictionsRF = modelRF\_Grid.transform(testData)  val lp = predictionsRF.select("indexed\_Th", "prediction")  val counttotal = predictionsRF.count()  val correct = lp.filter($"indexed\_Th"===$"prediction").count()  val wrong = lp.filter(not($"indexed\_Th"===$"prediction")).count()  val truep = lp.filter($"prediction"===0.0).filter($"indexed\_Th"===$"prediction").count()/counttotal.toDouble  val truen = lp.filter($"prediction"===1.0).filter($"indexed\_Th"===$"prediction").count()/counttotal.toDouble  val falsep = lp.filter($"prediction"===0.0).filter(not($"indexed\_Th"===$"prediction")).count()/counttotal.toDouble  val falsen = lp.filter($"prediction"===1.0).filter(not($"indexed\_Th"===$"prediction")).count()/counttotal.toDouble  val ratioWrong = wrong.toDouble/counttotal.toDouble  val ratioCorrect = correct.toDouble/counttotal.toDouble  println(ratioWrong)  println(ratioCorrect) |

## Lancement du programme en traitement de données massives via le cluster Dauphine ou AWS





### **Création du .jar**

Pour pouvoir lancer notre code scala sur le cluster Dauphine ou sur AWS, il faut au préalable créer un fichier .jar.

Ce fichier JAR (pour **J**ava **AR**chive) est une archive compressée (similaire à un zip) contenant un ensemble de données pour l’exécution de programme JAVA. Nous avons besoin de le créer car il va regrouper toutes les dépendances requises pour exécuter les travaux Spark sur un cluster YARN.

Plusieurs méthodes existent. Nous présentons ci-dessous la création du .jar avec SBT.

**Création du .jar avec SBT (Linux) – Méthode 1**

(Deux autres méthodes de génération du .JAR sont décrites en annexe)

Etape 1 : lancer sa machine virtuelle (pour être dans Linux)

Etape 2 : installer SDKMAN

Ouvrir un terminal et taper : $ curl –s https://get.sdkman.io | bash

Ouvrir un second terminal et taper : $ sdk version

Nous venons d’installer la version **SDK 5.7.4+362**

Etape 3 : installer SBT

Ouvrir un terminal et taper : $ sdk install sbt

Nous venons d’installer la version **sbt 1.3.6**

Etape 4 : vérifier la compatibilité des versions. Pour cela, taper :

$ spark-submit –version

$ spark-shell –version

$ spark-sql –version

Etape 5 : Il faut maintenant préparer un répertoire qui va accueillir le fichier .jar. Dans l’explorateur de fichiers, créer un répertoire et des sous-répertoires qui permettront à SBT de créer le .jar  voici ci-dessous l’architecture requise :

|  |
| --- |
| Dauphine  TP\_flights  |Projet001  |build.sbt  |src (répertoire) (où les fichiers sources ont été placés)  |main  |resources  |scala (répertoire) (où le code scala a été placé) |

Voici ce qui contient le fichier build.sbt :

|  |
| --- |
| name := "**flight**"  version := "1.0"  scalaVersion := "2.12.10"  libraryDependencies ++= Seq(  "org.apache.spark" %% "spark-sql" % "3.0.0-preview") |

Etape 6 : Nous allons maintenant générer le fichier .jar

Lancer un terminal et se déplacer dans le bon répertoire (endroit où se trouve le fichier build.sbt, qui est également celui de Projet001 dans notre cas, à l’aide des commandes cd et ls). Puis taper successivement les 3 lignes de code suivantes :

$ sbt

$ compile

$ package

Etape 7 : Le fichier .jar est à récupérer dans :

|  |
| --- |
| Dauphine  TP\_flights  |Projet001  |target (nouveau répertoire créé automatiquement)  |scala-2.12 (nouveau répertoire créé automatiquement)  | flight\_2.12-1.0.jar (nouveau fichier créé automatiquement)  | update (nouveau répertoire créé automatiquement)  | classes (nouveau répertoire créé automatiquement)  |streams (nouveau répertoire créé automatiquement)  |build.sbt  |src (répertoire) (où les fichiers sources ont été placés)  |main  |resources  |scala (répertoire) (où le code scala a été placé) |

### **Connection au cluster Dauphine**

Nous allons décrire ici plusieurs méthodes que nous avons utilisées pour nous connecter au cluster Dauphine.

**1. Connexion depuis la machine virtuelle linux**

Nous nous connectons au cluster Dauphine depuis notre machine Linux virtuelle.

Les commandes ls et cd nous permettent de nous déplacer dans les répertoires pour nous placer à l’endroit où se trouve le fichier fourni par Dario : « id\_rsa\_user160 ».

Nous devons ensuite utiliser ce fichier « id\_rsa\_user160 » pour nous identifier et nous connecter sur le cluster de Dauphine. Lorsqu’on est connecté, le prompt change.

|  |
| --- |
| (base) hyboui@UbuntuVM:~$ ls  anaconda3 Documents Téléchargements Dauphine Images  (base) hyboui@UbuntuVM:~$ cd Dauphine  (base) hyboui@UbuntuVM:~/Dauphine$ ls  id\_rsa\_user160  (base) hyboui@UbuntuVM:~/Dauphine$ chmod 600 id\_rsa\_user160  (base) hyboui@UbuntuVM:~/Dauphine$ ssh -i id\_rsa\_user160 -p 993 user160@www.lamsade.dauphine.fr |

**2. Connexion depuis l’application Ubuntu dans Windows**

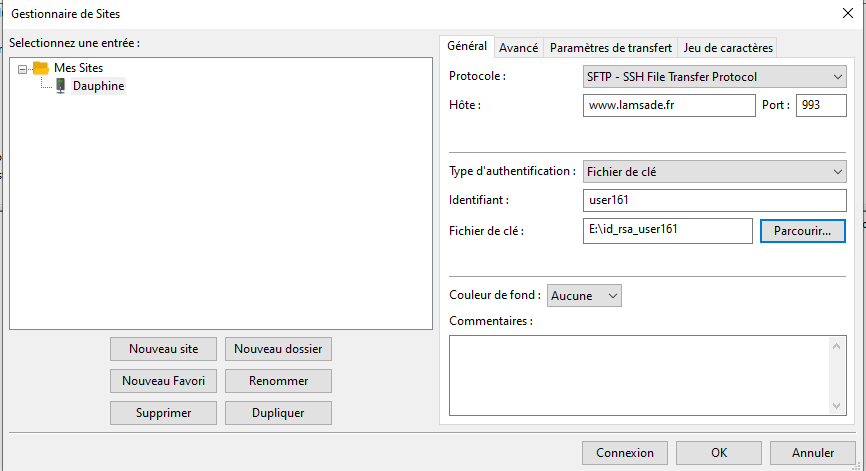
Au préalable, le fichier id\_rsa\_user160 a été mis dans un dropbox.Nous allons créer un fichier dédié au présent projet : hb/flightProject

A cet endroit-là, on charge la clef rsa.

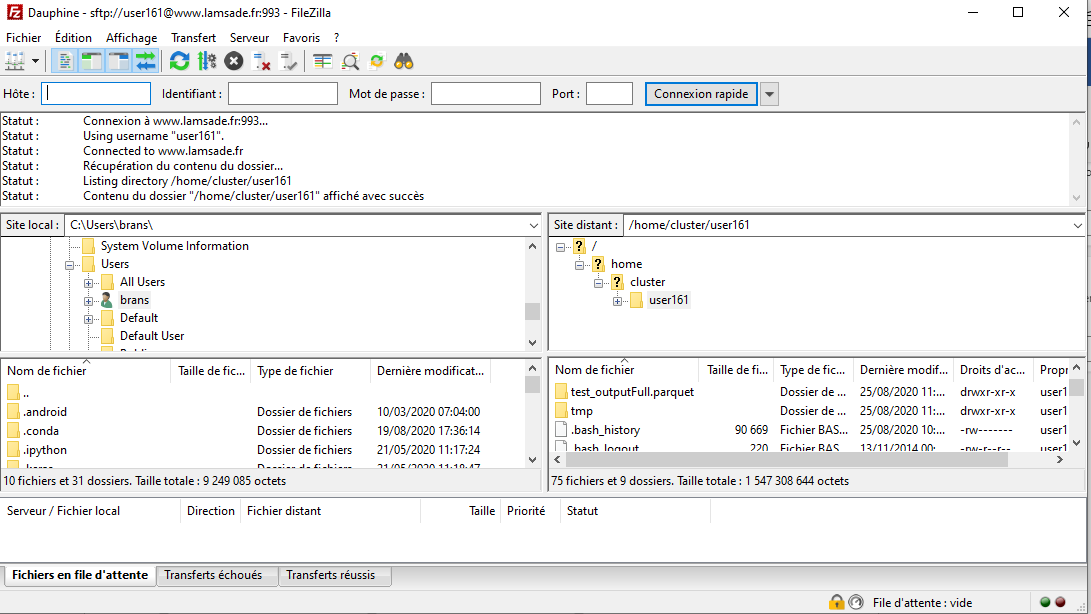
|  |
| --- |
| hyboui@LAPTOP-0CRTESPN:~$mkdir hb  hyboui@LAPTOP-0CRTESPN:~$ls  airflow airflow20200518 hb  hyboui@LAPTOP-0CRTESPN:~$cd hb  hyboui@LAPTOP-0CRTESPN:~/hb$mkdir flightProject  hyboui@LAPTOP-0CRTESPN:~/hb$ls  flightProject  hyboui@LAPTOP-0CRTESPN:~/hb$ cd flightProject  hyboui@LAPTOP-0CRTESPN:~/hb/flightProject$wget https://www.dropbox.com/s/ebjphj6do2xqxpg/id\_rsa\_user160  hyboui@LAPTOP-0CRTESPN:~/hb/flightProject$chmod 600 id\_rsa\_user160  hyboui@LAPTOP-0CRTESPN:~/hb/flightProject$ssh -i id\_rsa\_user160 -p 993 user160@www.lamsade.dauphine.fr |

**3. Connexion depuis une application gérant les connections sFTP (Filezilla)**

On utilise le paramétrage suivant pour se connecter au cluster de Dauphine en utilisant la clé privée :



Puis on accède au menu suivant :



Que ce soit la connexion depuis la machine virtuelle linux ou l’application Ubuntu de Windows, on doit obtenir :

|  |
| --- |
| \_\_ \_\_\_ \_\_ \_\_\_ | |\_\_ \_\_ \_ \_\_| | \_\_\_ \_\_\_ \_ \_\_ \_ \_\_ \_\_\_ \_\_ \_ \_\_\_| |\_ \_\_\_ \_ \_\_  \ \ / / '\_ ` \_ \| '\_ \ / \_` |/ \_` |/ \_ \ / \_ \| '\_ \| '\_ ` \_ \ / \_` / \_\_| \_\_|/ \_ \ '\_\_|  \ V /| | | | | | | | | (\_| | (\_| | (\_) | (\_) | |\_) | | | | | | (\_| \\_\_ \ |\_ | \_\_/ | \\_\_\_|\_|  \\_/ |\_| |\_| |\_|\_| |\_|\\_\_,\_|\\_\_,\_|\\_\_\_/ \\_\_\_/| .\_\_/|\_| |\_| |\_|\\_\_,\_|\_\_\_/\\_\_|  |\_|  Linux vmhadoopmaster 4.19.0-9-amd64 #1 SMP Debian 4.19.118-2 (2020-04-29) x86\_64  uptime : 22:06:44 up 30 days, 12:11, 1 user, load average: 0.00, 0.00, 0.00  Memoire utilisée : 5354 Mo / 63401 Mo (8%)  Free / Space : 9.9G / 18G  Free /opt/hdfs Space : 65G / 196G  Free /opt/local Space : 485G / 492G  Free /home/ccephfs Space : 2.0T / 4.9T  Free /home/cluster Space : 155G / 1.6T  ens3 adress : 10.40.178.80  Last login: Sun Jul 12 20:03:28 2020 from 193.48.71.250  user160@vmhadoopmaster:~$ hdfs dfs -ls /user/user160  Found 14 items  drwxr-xr-x - user160 cluster 0 2019-12-03 21:46 /user/user160/.sparkStaging  -rw-r--r-- 3 user160 cluster 1428843 2019-06-18 14:48 /user/user160/5000-8.txt  -rw-r--r-- 3 user160 cluster 411812 2019-09-09 19:12 /user/user160/R1\_users.csv  -rw-r--r-- 3 user160 cluster 946079 2019-09-09 19:12 /user/user160/R2\_books.csv  -rw-r--r-- 3 user160 cluster 3608934 2019-09-09 19:12 /user/user160/S\_ratings.csv  -rw-r--r-- 3 user160 cluster 789 2019-09-20 07:40 /user/user160/VeillerTard.txt  -rw-r--r-- 3 user160 cluster 20268 2019-09-30 18:59 /user/user160/derby.log  drwxr-xr-x - user160 cluster 0 2020-07-09 21:01 /user/user160/flight  -rw-r--r-- 3 user160 cluster 4550 2019-11-30 21:17 /user/user160/iris.data.txt  -rw-r--r-- 3 user160 cluster 244 2019-09-09 19:12 /user/user160/keywords.csv  drwxr-xr-x - user160 cluster 0 2019-09-30 18:59 /user/user160/metastore\_db  drwxr-xr-x - user160 cluster 0 2019-06-18 15:18 /user/user160/outputspark  -rw-r--r-- 3 user160 cluster 56295113 2019-09-30 18:59 /user/user160/vk\_001.json  -rw-r--r-- 3 user160 cluster 111993299 2019-09-30 19:00 /user/user160/vk\_002.json |

## Vérification des versions Spark et Scala du cluster Dauphine

Une fois connecté au cluster Dauphine, on peut aller regarder la version utilisée pour scala et spark. Ces informations sont utiles pour le fichier build.sbt :

user160@vmhadoopmaster:~$spark-submit –version

Dans notre cas, nous avons :

* spark version 3.0.0-Preview2
* Scala version 2.12.10

### **Chargement des fichiers (sources et .jar) dans HDFS**

*Remarque : Le répertoire hdfs dans le cluster Dauphine à l’emplacement* ***/user/user160*** *est le seul endroit où nous sommes autorisés à utiliser à notre guise : création de fichiers et sous répertoires pour nos besoins.*

Etape 1 :

Nous avons au préalable placé les fichiers sources suivantes, ainsi que le fichier .jar dans un dropbox :

* flight\_1M.csv
* flight\_6M.csv
* flight\_12M.csv
* flight\_24M.csv
* weather\_1M.csv
* weather\_6M.csv
* weather\_12M.csv
* weather\_24M.csv
* flight\_2.12-1.0.jar

Etape 2 :

Pour charger les fichiers dans ce répertoire qui nous est dédié, on commence par les charger dans le cluster, avant de les pousser dans hdfs.

Lancer la machine virtuelle et se connecter au cluster Dauphine :

|  |
| --- |
| (base) hyboui@UbuntuVM:~$ ls  (base) hyboui@UbuntuVM:~$ cd Dauphine  (base) hyboui@UbuntuVM:~/Dauphine$ ls  (base) hyboui@UbuntuVM:~/Dauphine$ chmod 600 id\_rsa\_user160  (base) hyboui@UbuntuVM:~/Dauphine$ ssh -i id\_rsa\_user160 -p 993 user160@www.lamsade.dauphine.fr  user160@vmhadoopmaste:~$hdfs dfs -ls /user/user160 |

Charger en local et mettre une copie dans hdfs :

|  |
| --- |
| user160@vmhadoopmaste:~$ hdfs dfs -mkdir/user/user160/flight  user160@vmhadoopmaste:~$wget https://www.dropbox.com/s/qz08465r7hxg9d3/Flight\_Clean\_Data\_V2.zip  user160@vmhadoopmaste:~$unzip Flight\_Clean\_Data\_V2.zip  user160@vmhadoopmaste:~$mv Flight\_Clean\_Data/\* ../flight  user160@vmhadoopmaste:~$ hdfs dfs -put flight\_24M.csv /user/user160/flight  user160@vmhadoopmaste:~$wget https://www.dropbox.com/s/ymgo53ye2q5q3hq/flight\_2.12-1.0.jar  user160@vmhadoopmaste:~$hdfs dfs -ls flight |

*Remarque :*

*Pour afficher le contenu d’un fichier en local, taper : $cat xxx.csv*

*Pour afficher le contenu d’un fichier sur hdfs, taper $hdfs dfs -cat /user/user160/xxx.csv*

### **Lancement des job Spark en local, Stand Alone et sur Yarn**

**Rappel sur la configuration du cluster Dauphine**

Il nous a été très difficile de connaître la configuration exacte mise à disposition. Voici les seules informations dont nous sommes à peu près sûrs :

* Le cluster Dauphine est composé de 9 nœuds
* 213 GB de mémoire disponible

**Les adaptations entre le lancement en local, en Stand Alone ou sur YARN**

Nous avons lancé notre programme à la fois en local, mais aussi sur Yarn. Nous voulions comparer les différences de comportements.

|  |  |  |
| --- | --- | --- |
| Lancement en local | Lancement en Stand Alone  (ressources allouées par Spark) | Lancement sur Yarn |
| Le code .scala (avant la génération du .jar) :  val spark = SparkSession  .builder  .appName("predict\_delay")  .getOrCreate() | | |
| Modification de la commande spark-submit :  spark-submit --class flight --master local[8] flight\_2.12-1.0.jar | Modification de la commande spark-submit :  spark-submit  --class flight  **--master spark://10.40.178.80:7077**  --executor-cores 4  --num-executors 12  --executor-memory 5g  --conf spark.executor.memoryOverhead=2g  --conf spark.driver.memory=5g  --conf spark.driver.cores=4  flight\_2.12-1.0.jar | Modification de la commande spark-submit :  spark-submit  --class flight  **--master yarn**  **--deploy-mode cluster**  --executor-cores 4  --num-executors 12  --executor-memory 4g  --conf spark.executor.memoryOverhead=2g  --conf spark.driver.memory=5g  --conf spark.driver.cores=2  flight\_2.12-1.0.jar |

Remarque : Les codes ci-dessous fonctionnent sur le programme de « pipeline et modèle », mais aussi sur le programme de « jointure » où il suffit d’adapter le fichier .jar.

Pour connaitre l’adresse ip pour le lancement en local, nous avons utilisé la commande  /sbin/ifconfig, puis récupérer l’adresse INET.

### **Tentatives d’optimisation**

Optimisations via les paramètres de la commande spark-submit :

Avec la commande spark-submit, il est possible d’adapter des variables tels que *num\_executors, executor\_core, executor-memory, spark.executor.memoryOverhead, spark.driver.memory et spark.driver.cores*.

De même, nous avons optimisé la gestion des fichiers temporaires (pour l’exécution en stand alone pour les problèmes de mémoire)

--conf spark.local.dir=/home/cluster/user160/flight/tmp

Voici quelques liens qui nous ont permis de mieux comprendre le fonctionnement et le choix des valeurs des paramètres :

<https://spark.apache.org/docs/latest/cluster-overview.html>

<https://spark.apache.org/docs/latest/configuration.html>

Sur Yarn, 2 modes de déploiements sont possibles :

* Mode cluster : où tout le job s'exécute dans le cluster, c'est à dire les Spark Executors (qui exécutent les vraies tâches) et le Spark Driver (qui ordonnance les Executors). Ce dernier sera encapsulé dans un YARN Application Master.
* Mode client : où Spark Driver s'exécute sur la machine cliente (tel que votre propre ordinateur portable). Si votre machine s'éteint, le job s'arrête. Ce mode est approprié pour les jobs interactifs.

Nous avons uniquement utilisé le mode cluster.

Dans le code en scala, nous avons également essayé l’ajout des lignes de code qui optimisent le traitement tels que des « .persist(StorageLevel.MEMORY\_AND\_DISK\_SER) » et « .unpersist()»sur les dataframes.

Tentative d’optimisation avancées via d’autres paramètres :

Allocation dynamique des ressources :

--conf spark.dynamicAllocation.minExecutors=4

--conf spark.dynamicAllocation.maxExecutors=48

--conf spark.dynamicAllocation.initialExecutors=12

Optimisation au niveau du garbage collector

--conf "spark.driver.extraJavaOptions=-XX:+UseG1GC"

--conf "spark.executor.extraJavaOptions=-XX:+UseG1GC"

Sérialisation KRYO (dans le code Scala) :

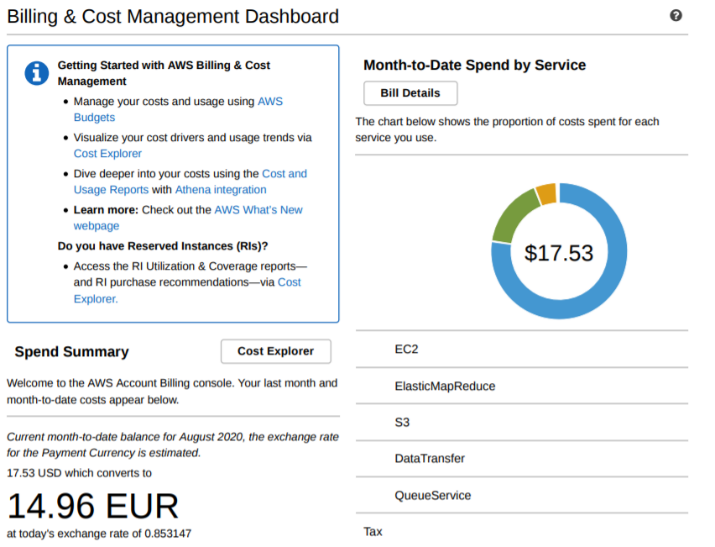
.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

.config("spark.kryo.registrationRequired", false)

### **Conclusion sur les clusters Dauphine et AWS :**

**CLUSTER DAUPHINE** : Sur le cluster Dauphine, nos tentatives ont été infructueuses. Le programme intégrant le GridSearch (données basées sur le dataset1 vol, le plus volumineux), le programme a tourné pendant plusieurs jours ( ! ).

**ROSETTA HUB** : nous avons atteint la limite de la subvention et nous n’avons pu tester sur AWS via Rosetta Hub.

**AWS** : Nous avons pensé lancer ces mêmes codes sur AWS (basique sans optimisation avancées), nos tentatives ont été restreintes car l’utilisation d’AWS est onéreuse et les résultats n’ont pas été meilleurs. Notre unique tentative a duré 6h avant de planter.

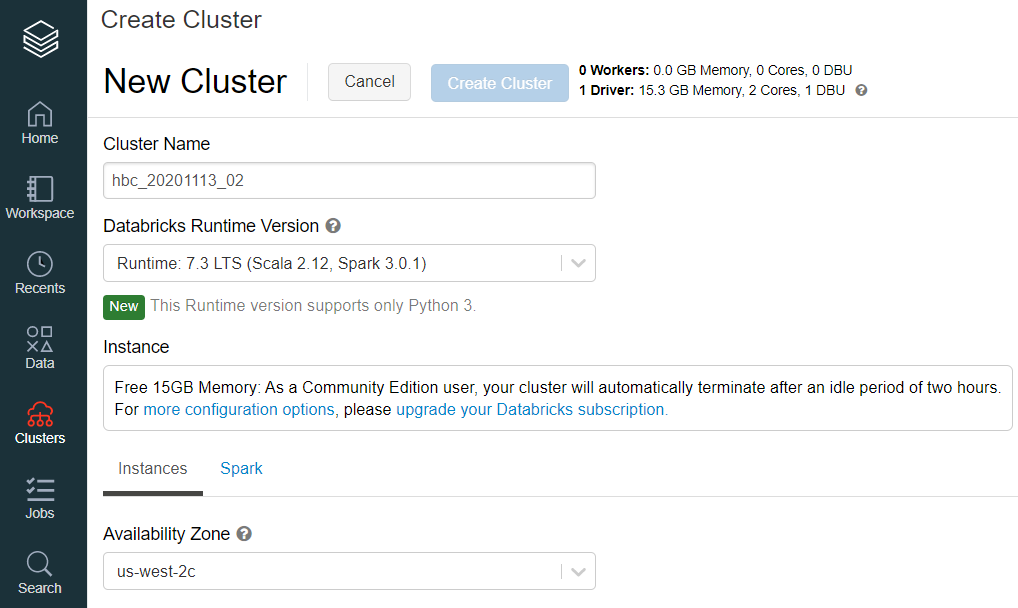
* Nos données ont été hébergées sur S3.
* Le code est sur EMR
* EC2 nous donne les clefs de connexion

## Traitement de données massives via DATABRICKS

Face aux nombreuses difficultés rencontrées sur le cluster Dauphine, nous avons décidé d’utiliser DATABRICKS (version community).

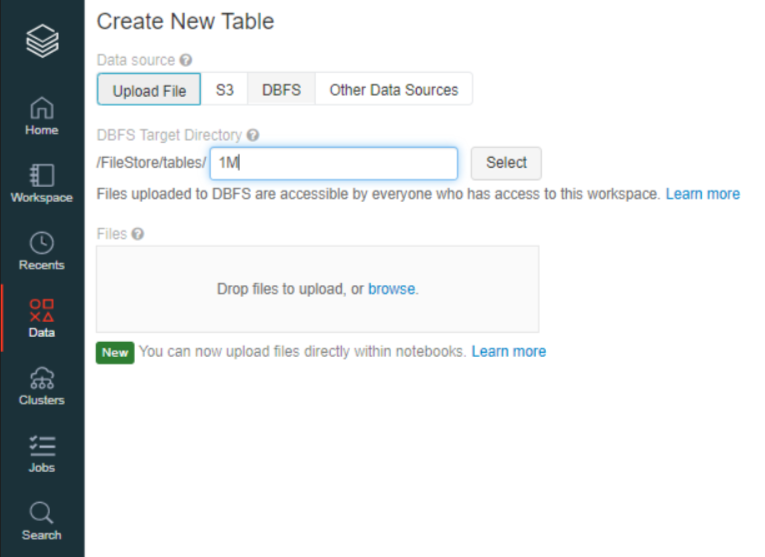
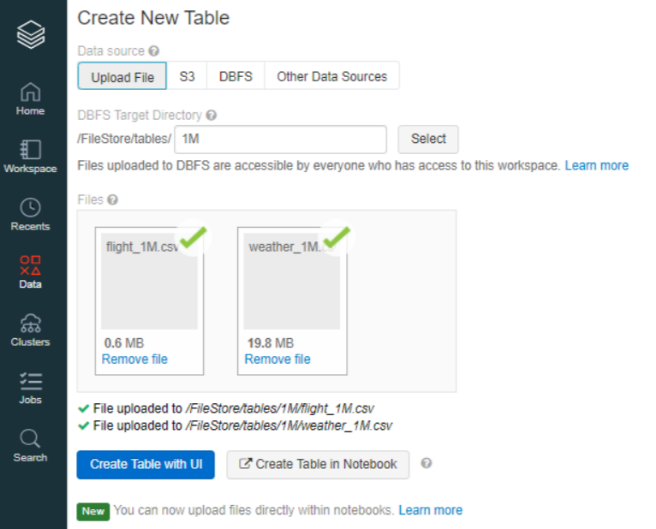
* Interruption, indisponibilité, désactivation du compte étudiant (user160) du cluster
* Complexité de partage des données entre les membres du projet (contournée par l’utilisation du même compte),
* Absence de GUI, absence de Spark UI pour analyser les “jobs”, “stages” et les “tasks”

Etape 1 : Création d’un cluster



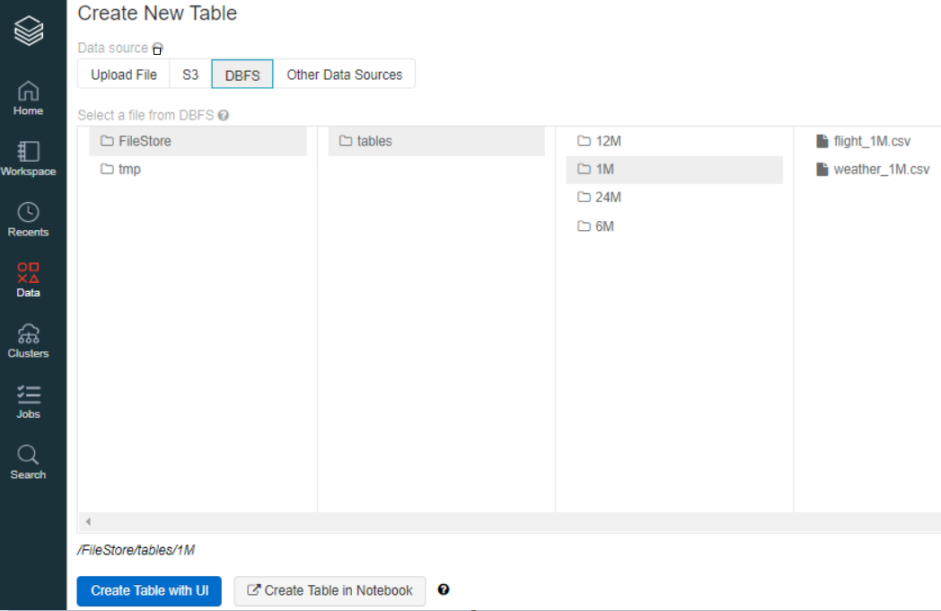
Etape 2 : Création des répertoires sur FileStore / Transfert des données CSV

Création du répertoire dans FileStore : Dépôt des fichiers :

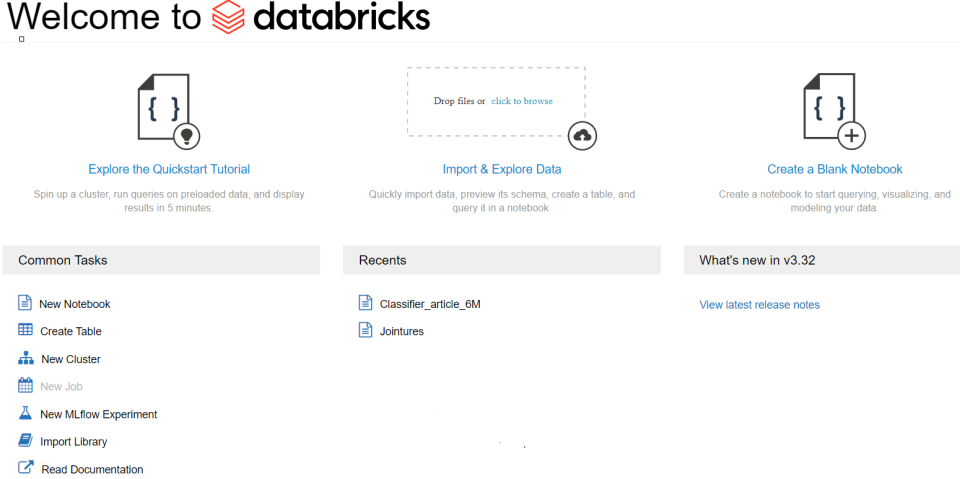


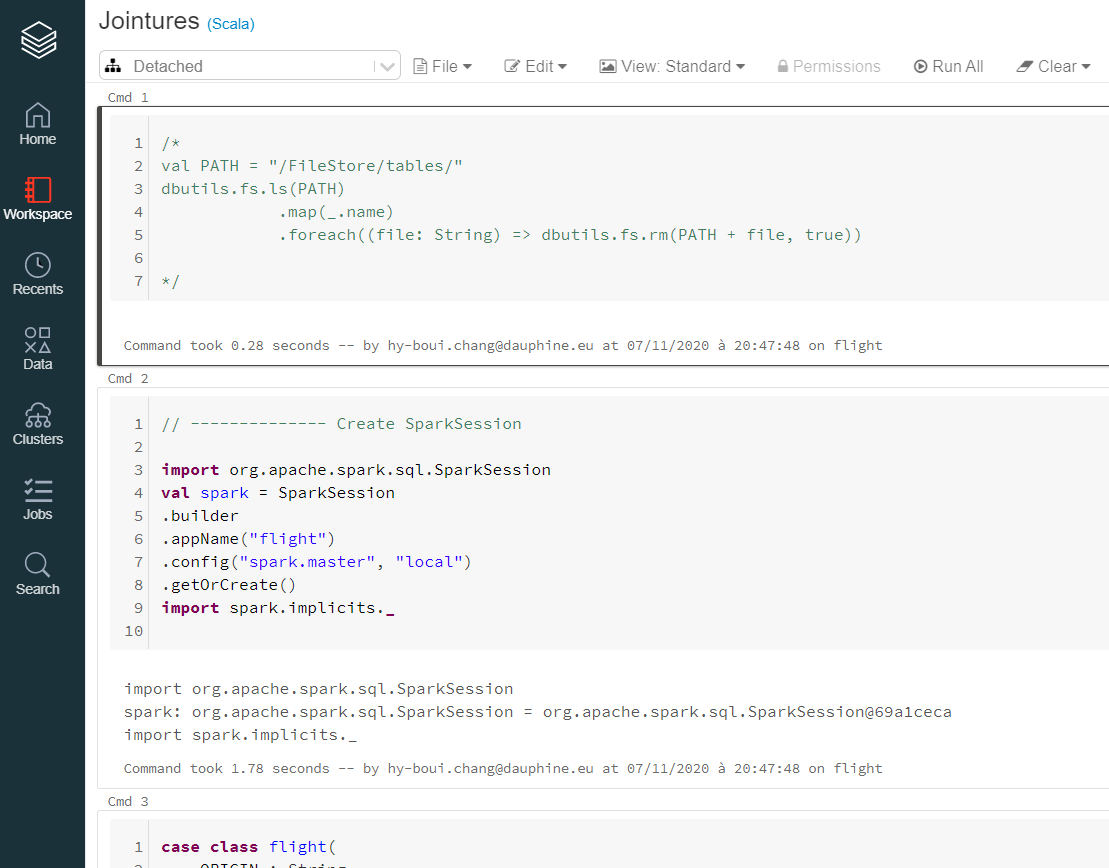
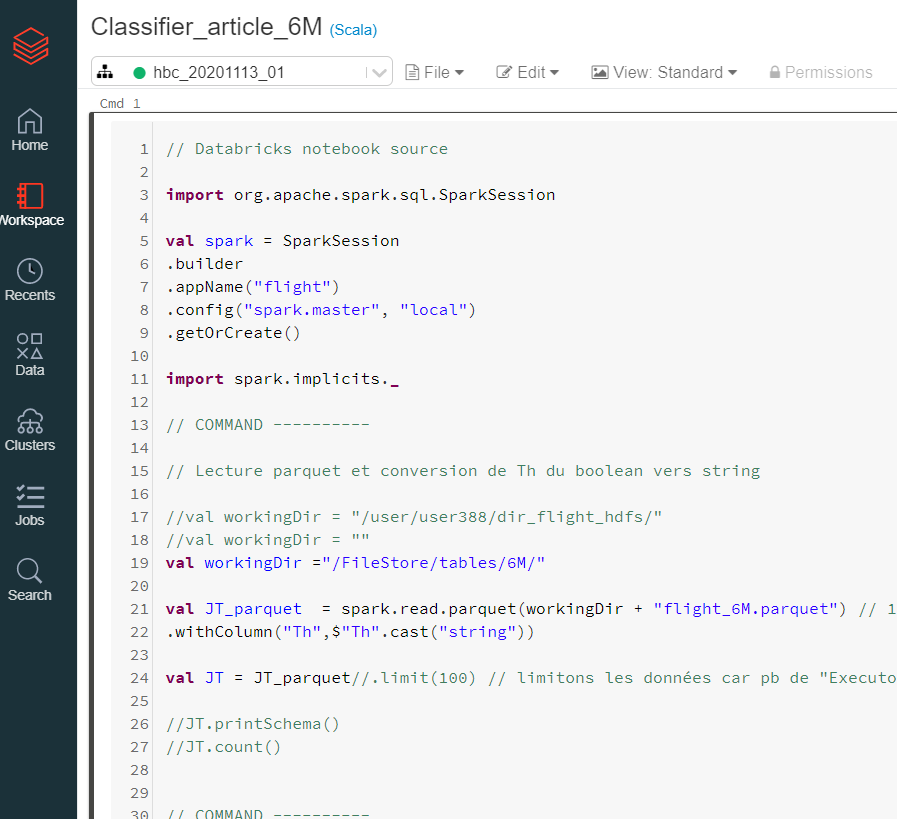
Voici la capture d’écran de nos données de départ :

Note : seuls les fichiers Parquet seront à conserver à la fin car nous avons dû supprimer les fichiers CSV pour éviter d’atteindre la limite de stockage allouée en version community.



Etape 3 : Création des notebooks en scala pour les jointures et le pipeline/modèle







Etape 4 : Lancement des programmes scala

Avec cette version community, le cluster mis à disposition s’éteint au bout de 2h d’inactivité. Il est donc nécessaire de s’y connecter régulièrement.

# Résultats

Nous avons testé sur 1 mois, 6 mois, 12 mois et 24 mois.

Le modèle utilisé est le RANDOM FOREST.

Les données ont été rééquilibrées (50% vols retardés, 50% vols à l’heure).

## Résultats sur les deux versions de datasets vol que nous avons testé

### **Résultats avec le premier dataset1 de vol**

|  |  |
| --- | --- |
| Données sur **1 MOIS** | 12 slots (0h à -12h) avec colonnes S & D |
| Temps de calcul | Random Forest |
| Accuracy | 60% |

|  |  |  |
| --- | --- | --- |
| Données sur 2 **MOIS** | 12 slots (0h à -12h)  Avec colonnes S & D | 12 slots (0h à -12h) Avec colonnes S & D |
| Modèle | Random Forest | Gradient Boosted Trees |
| Accuracy | 59,7% | 62,4% |

### **Résultats avec le second dataset2 de vol**

|  |  |  |  |
| --- | --- | --- | --- |
| Données sur **1 MOIS** | 3 slots (0h, -6h, -12h)  Sans colonne S & D | 3 slots (0h, -6h, -12h)  & encodage colonne S | 7 slots (0h à -6h)  Sans colonne S & D |
| Temps de calcul (databricks) |  | 20 minutes | 22 minutes |
| Paramètres opti males | Max Bins = 200  Max Depth = 20  Num Trees = 200  Impurity = Gini | Max Bins = 200  Max Depth = 20  Num Trees = 200  Impurity = Gini | Max Bins = 200  Max Depth = 20  Num Trees = 200  Impurity = Gini |
| Training Accuracy | 99.97% | NA | NA |
| Test Accuracy | 84,65% | **84,7%** | 82,8% |
| Training F1 score | 99.97% | NA | NA |
| Test F1 score | 84.65% | NA | NA |

|  |  |  |  |
| --- | --- | --- | --- |
| Données sur **6 MOIS** | 3 slots (0h, -6h, -12h) Sans colonne S & D | 3 slots (0h, -6h, -12h)  & encodage colonne S | 7 slots (0h à -6h) Sans colonne S & D |
| Temps de calcul (databricks) |  | 2h 22 minutes | 1h40 |
| Paramètres optimales | Max Bins = 200  Max Depth = 25  Num Trees = 200  Impurity = Entropy | Max Bins = 200  Max Depth = 25  Num Trees = 200  Impurity = Entropy | Max Bins = 200  Max Depth = 25  Num Trees = 200  Impurity = Entropy |
| Accuracy | **79,3%** | 79,2% | 79,2% |

|  |  |  |  |
| --- | --- | --- | --- |
| Données sur **12 MOIS** | 3 slots (0h, -6h, -12h) Sans colonne S & D | 3 slots (0h, -6h, -12h)  & encodage colonne S | 7 slots (0h à -6h) Sans colonne S & D |
| Temps de calcul (databricks) |  | 4h | 3h |
| Paramètres optimales | Max Bins = 200  Max Depth = 15  Num Trees = 200  Impurity = Entropy | Max Bins = 200  Max Depth = 20  Num Trees = 200  Impurity = Entropy | Max Bins = 200  Max Depth = 20  Num Trees = 200  Impurity = Entropy |
| Accuracy | 77,5% | 78,4% | **78,6%** |

|  |  |  |  |
| --- | --- | --- | --- |
| Données sur **24 MOIS** | 3 slots (0h, -6h, -12h) Sans colonne S & D | 3 slots (0h, -6h, -12h)& encodage colonne S | 7 slots (0h à -6h) Sans colonne S & D |
| Temps de calcul (databricks) | 6h | 8h | 8h |
| Paramètres optimales | Max Bins = 200  Max Depth = 25  Num Trees = 150  Impurity = Entropy | Max Bins = 200  Max Depth = 25  Num Trees = 200  Impurity = Entropy | Max Bins = 200  Max Depth = 25  Num Trees = 200  Impurity = Entropy |
| Accuracy | 78,6 % | **78,7%** | 78,6 % |

Nous avons uniquement regardé la valeur de l’accuracy car nos données sont équilibrées (50% de vols en retard, 50% de vols à l’heure).

## Interprétation des résultats

Les résultats obtenus sont assez proches de ceux publiés dans l’article malgré une limitation des ressources matériels dont nous disposions.

Nous notons également que notre modèle overfitte sur 1 mois, cependant, par manque de ressources disponibles, nous n’avons pas pu faire l’analyse sur les périodes plus longues. Sur les données de 1 mois, nous avons modifié les paramètres suivants à la baisse (le nombre d’arbres, le MaxDepth), et d’autres à la hausse (le nbre d’instance minimal) pour tenter d’accroître la capacité de génération de notre modèle. Malheureusement, cela impacte également négativement l’accuracy des données de test.

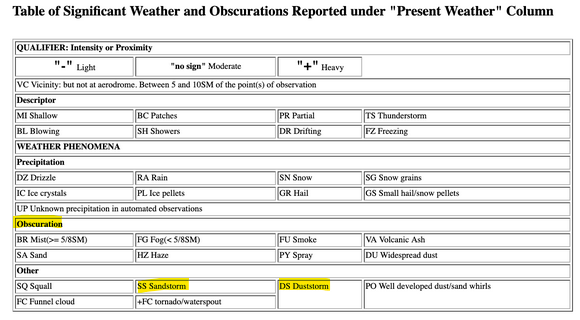
De plus, on observe que malgré l’augmentation des données (+ de mois), notre accuracy diminue, ce qui est contre-intuitif. Nous continuons à investiguer sur les explications probables, notamment en s’appuyant sur l’article suivant : <https://www.kdnuggets.com/2015/06/machine-learning-more-data-better-algorithms.html>.

Compte tenu du fait que la performance sur 1 mois soit supérieure (janvier 2012), il faudrait vérifier si le mois en question n’était pas un mois inhabituel en termes de météo (épisode durable de tempêtes de neige etc. par exemple), ce qui pourrait expliquer cette précision plus élevée que sur d’autres périodes.

Nous avons également pensé faire varier l’hyperparamètre THRESHOLD pour améliorer les performances de notre modèle. Cependant, cet hyperparamètre n’est modifiable que pour les classifieurs multi-classes sous ML Spark.

Concernant les champs complexes de Sky Conditions et Weather Conditions :

* On peut observer que l’application d’un encodage spécifique sur la colonne S (Sky Condition) sans l’utilisation de la fonction StringIndexer), semble améliorer très légèrement l’accuracy.
* De façon similaire, l’encodage de la colonne D (Weather Condition) a été testé mais n’a pas amélioré le modèle, même en prenant en compte que les phénomènes extrêmes (Tempêtes et phénomènes météorologiques obscurcissants).



En termes de comparabilité, il faut également prêter attention à l’impact possible de la saisonnalité sur l’homogénéité des données. Et donc, plutôt comparer le mois de Janvier 2012 vs 2013 ou l’évolution des résultats sur période de 12 mois vs 24 par exemple.

## Pour aller plus loin …

Tester d’autres modèles:

Nous n’avons pas pu profiter du GridSearch sur 24 mois, même sur DATABRICKS. Nous aurions aimé pouvoir tester d’autres modèles tels que le Gradient Boosted Tree, le LIGHT GBM ou le XGBOOST 4J, etc

|  |  |  |
| --- | --- | --- |
| Modèles | Lien vers le Jupyter Notebook | Lien vers le fichier .scala |
| RANDOM FOREST CLASSIFIER |  |  |
| LOGISTIC REGRESSION |  |  |
| GRADIENT BOOSTED TREE CLASSIFIER |  |  |

Tester d’autres scenarios en nbre de slots (créneaux horaires différents) :

Dans la majorité des cas, nous sommes partis sur la prise en compte de 6 créneaux météorologiques d’une heure avant le départ et l’arrivée. Car 12 créneaux fait beaucoup de données pour l’apprentissage du modèle. Cela peut nuire à l’efficacité de l’apprentissage.

On pourrait envisager d'ajuster/diminuer le nombre de créneaux horaires. Cela aurait aussi tendance à diminuer le temps de calcul pour effectuer la jointure entre la table des vols et la table des données météo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Scénario 1  Uniquement les données météo de l’aéroport de départ, à l’heure de départ prévu | Scénario 2  Uniquement les données météo de l’aéroport d’arrivée, à l’heure d’arrivée prévu | Scénario 3  Les données météo de l’aéroport de départ, à l’heure de départ prévu  Les données météo de l’aéroport d’arrivée, à l’heure d’arrivée prévu | Scénario 4  Les données météo de l’aéroport de départ, à l’heure de départ prévu + jusqu’à 3 heures avant  Les données météo de l’aéroport d’arrivée, à l’heure d’arrivée prévu + jusqu’à 3 heures avant | Scénario 5  Les données météo de l’aéroport de départ, à l’heure de départ prévu + jusqu’à 6 heures avant  Les données météo de l’aéroport d’arrivée, à l’heure d’arrivée prévu + jusqu’à 6 heures avant | Scénario 6  Les données météo de l’aéroport de départ, à l’heure de départ prévu + jusqu’à 12 heures avant  Les données météo de l’aéroport d’arrivée, à l’heure d’arrivée prévu + jusqu’à 12 heures avant |

Optimiser l’encodage spécifique des colonnes SkyCondition (S) et Descriptions (D)

Mieux appréhender l’interprétation et l’importance des colonnes S (Sky Conditions) et D (weather description) avec un expert métier (météorologue, contrôleur aérien) pour déterminer quelles sont les informations météorologiques majeures qui influent sur les retards, et encoder que celles-là de façon plus optimale.

# Conclusions sur le travail demandé

## Les difficultés rencontrées

### **Incertitudes quant à l’architecture précise du cluster Dauphine**

L’université nous a mis à disposition le cluster Dauphine avec très peu d’information sur l’architecture du cluster, et nous en avions besoin pour comprendre les problèmes techniques rencontrés lors du lancement des jobs.

Nous aurions aimé la mise en place d’un référent / support technique pour nous accompagner. Ceci a été mis en place tardivement pour la réalisation du projet.

Face à ces difficultés, nous avons également testé notre programme sur **AWS**. Nous avons ouvert un compte afin de voir si nous rencontrions les mêmes difficultés que sur le cluster Dauphine. L’expérience (payante) ne fut pas concluante.

Le Cluster de Dauphine n’étant soudainement plus accessible depuis le début du mois d’octobre, par voie de conséquence nous n’avons plus accès aux logs des exécutions sur le cluster Dauphine...

Nous avons dû migrer les données vers la plate-forme “Databricks Community” pour continuer certains tests. Ce qui nous a pris du temps supplémentaire.

### **Organisation et gestion de projets**

Compte tenu de l’importance de la charge de travail pour ce projet en équipe, cela demande de l’expérience en termes de gestion de projet afin d’organiser au mieux l’ensemble des tâches.

Cet aspect n’a pas été abordé en cours mais cela a son influence lorsque l’on veut optimiser la gestion du projet, les versioning, les phases de beta test, d’exploration et de mise en production globale d’un pipeline complet.

### **Temps d’exécution global (plusieurs dizaines heures voire de jours)**

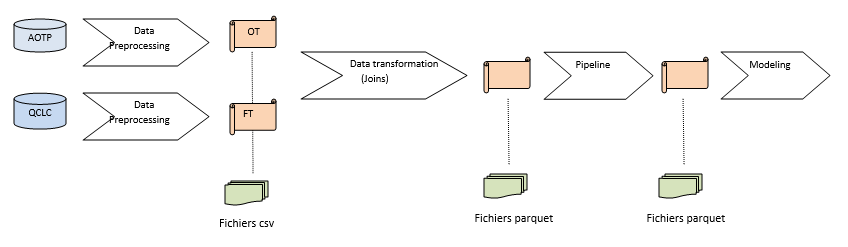
L’accès au journal Spark nécessite de laisser sa session au Cluster Dauphine ouverte tout le temps de l’exécution du programme. Toute rupture de connexion, oblige à relancer le programme.

Le travail est extrêmement chronophage, difficile à concilier avec nos journées au travail avec employeurs respectifs.

La solution serait de pouvoir lancer le WE mais souvent il y a un problème de bande passante (nous ne sommes pas les seuls) et 48h ne suffisent pas.

Action mise en œuvre :

* Découpage du processus global en sous- processus avec des fichiers sauvegardés sur disque (.csv ou . parquet) ce qui évite de perdre du temps en relançant l’ensemble des traitements. La conséquence est de « découper » l’ensemble du programme en sous-programmes séquentiels.



* Parallélisation des pistes de débogage entre membres de l’équipe projet. Chacun teste des solutions via son compte Cluster Dauphine.

Cependant le temps de transfert de fichiers (upload) est très important (5h pour 2Go en connexion ADSL classique) : ce qui rend l’échange de données peu pratique entre membres de l’équipe projet.

### **Log d’erreur Spark peu ou pas explicite**

Nous avons eu des erreurs du type « Container exited with a non-zero exit code 13 » ou « java.io.IOException: No space left on device »

Actions mises en œuvre :

* Insertion lignes d’impression (println) afin de localiser le point d’arrêt du programme.
* Insertion de lignes de codes de type « show » ou « count » pour cerner les points d’arrêts et contourner les problématiques de calcul sur ordre « Lazy evaluation ».

## Ce que nous avons appris

Les leçons que nous en avons tiré sont :

* Les difficultés rencontrées ont été des occasions d’apprendre et d’approfondir les sujets abordés en cours. Cela a été riche en expériences qui, à coup sûr, nous serons utiles dans le monde du travail.
* Il n’est pas possible de traiter autant de données sur son propre ordinateur. Il est donc nécessaire de passer par une solution de partitionnement et parallélisation, type Spark, pour ce genre d’analyse.
* Ce n’est pas parce que le programme sur l’ordinateur fonctionne, que la génération en .JAR se passe sans problème. Il faut souvent débuguer pour réussir la génération du .JAR.
* Ce n’est pas parce que le programme fonctionne parfaitement sur son ordinateur en local (Jupyter notebook, par exemple) qu’il fonctionnera sur le cluster.
* Nous avons dû chercher et recourir à différentes techniques pour optimiser notre programme face à la quantité de données.
* Un job qui est lancé et qui tourne pendant plusieurs heures peut s’arrêter brusquement sans que le journal (log) fournisse vraiment d’explications.
* Il est certes important de savoir manipuler des données massives grâce à la programmation et l’architecture technique, mais il faut également bien analyser et comprendre les données sous-jacentes afin de les sélectionner et les mettre en forme de façon judicieuse, en découle des différences de performance et de ressources nécessaires.

# Annexe 1a – Création des .JAR avec SBT sous Linux

Etape 1 : Créer un répertoire qui va accueillir le .jar. Appelons le Directory001

Dans ce répertoire, mettre un seul fichier scala : notre programme

Dans ce répertoire, mettre également le fichier build.sbt

Dans ce répertoire, créer un sous répertoire nommé « project ». Là on va y mettre le fichier assembly.sbt

|  |
| --- |
| Dauphine  TP\_flights  |Directory001  Programme.scala  |build.sbt  |project (répertoire)  Assembly.sbt |

Voici le fichier build.sbt :

|  |
| --- |
| name := "essai"  version := "1.0"  scalaVersion := "2.12.10"  libraryDependencies ++= Seq(  "org.apache.spark" %% "spark-sql" % "3.0.0-preview",  "org.apache.spark" %% "spark-core" % "3.0.0-preview"  )  javaOptions += "-Xmx2g"  javaOptions += "-XX:ReservedCodeCacheSize=1g" |

Voici le fichier assembly.sbt :

|  |
| --- |
| addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.6") |

Etape 2 : Nous allons maintenant générer le fichier .jar

Lancer un terminal et se déplacer dans le bon répertoire (endroit où se trouve le fichier build.sbt, qui est également celui de Directory dans notre cas, à l’aide des commandes cd et ls). Puis taper successivement les 2 lignes de code suivantes :

$ sbt assembly

$ sbt package

Etape 3 : Le fichier .jar est à récupérer dans :

|  |
| --- |
| Dauphine  TP\_flights  |Directory001  Programme.scala  |build.sbt  |project (répertoire)  Assembly.sbt  |target (nouveau répertoire créé automatiquement)  |scala-2.12 (nouveau répertoire créé automatiquement)  | essai\_2.12-1.0.jar (nouveau fichier créé automatiquement)  | update (nouveau répertoire créé automatiquement)  | classes (nouveau répertoire créé automatiquement)  | test-classes (nouveau répertoire créé automatiquement)  |streams (nouveau répertoire créé automatiquement) |

Le .jar se trouve ensuite dans le répertoire "target/scala-2.X"

Pour info je t'ai mis mon .build, assembly.sbt et j'ai installé sbt 1.3.13

# Annexe 1b – Création des .JAR avec IntelliJ sous Windows

Etape 1 : Installation de INTELLIJ IDEA version *Community*

Etape 2 : Téléchargement le plugin SCALA :  *File -> Settings..*. Depuis cette fenêtre, cliquez sur « *plugins* » dans la colonne de gauche puis recherchez le plugin « *scala* ». Cliquez sur Installer

Etape 3 : Création d’un nouveau projet : *File -> new -> Project*. Dans la colonne de gauche, sélectionnez SCALA, et dans la partie de droite, sélectionner SBT.

Appelez ce projet « **flight** », qui doit être le même nom que celui qui se trouve dans le fichier build.sbt.

Un projet va se créer avec l’arborescence des répertoires qui convient.

Etape 4 : Intégration du projet en scala. Allez dans le répertoire flight/src/main/scala. Ce répertoire est en couleur bleu. Cliquez droit dessus et choisir new/Scala class. Donnez un titre à votre fichier scala (par exemple **builderFlight**) et choisir le type « Object ».

Ce fichier builderFlight.scala de type Object doit être codée de la façon suivante :

Toutes les librairies d’import : en dehors de l’object.

Dans l’object, mettre un main {} et dans le main, le code scala en entier.

|  |
| --- |
| Import xxxx librairies xxxx  object flightProject {  def main (args : Array[String]) : Unit = {  xxxxxx code scala xxxx  }  } |

*Remarque : Dans le code scala, mettre le chemin vers les bons répertoires hdfs (vol et météo) où nous mettrons les fichiers sources : val workingDirectory = « hdfs:/user/user160/ »*

*Dans notre cas, les 24 mois sont des fichiers météo et vol différents  dans hdfs, créer un répertoire pour accueillir les 24 fichiers vol, et un autre répertoire pour accueillir les 24 fichiers météo. Le programme va tout concaténer en 2 années complètes.*

Etape 5 : le fichier build.sbt est à adapter en fonction des versions scala que nous allons utiliser. Pour connaitre la bonne version, une fois connecté au cluster Dauphine, taper :

user160@vmhadoopmaster:~$spark-submit –version

Etape 6 : Génération du .jar

Pour cela, ouvrir le terminal en bas de IntelliJ. Taper « sbt package »

On récupère le fichier .jar dans l’arborescence :

flight/target/scala-2.13/update