databricksPhase4_Submission_Final

Predicting Flight Delays Two Hours in Advance

Section 5 Group 3 Final Project

Summer 2022: Friday 2pm Section with Vinicio De Sola

Alex Carite, Muhammad Jawaid, Peter Morgan, and Thomas Welsh

Team Introduction

Team Members

Alex Carite (alex.carite@berkeley.edu (mailto:alex.carite@berkeley.edu)) worked closely with weather data and extracting more information from the REM field. Helped with report writing and early EDA of airplane and weather datasets. Worked with blob storage and the saving of dataframes to parquet files. Prepared logistic regression and XGBoost models and building out the machine learning pipeline (including feature scaling, one hot encoding, and imputations).

Muhammad Ali Jawaid (mjawaid@berkeley.edu (mailto:mjawaid@berkeley.edu)) worked with team on establishing model pipeline, exploratory data analysis, and data munging to facilitate join. Worked on feature engineering and machine learning models.

Peter Morgan (peter_morgan@ischool.berkeley.edu (mailto:peter_morgan@ischool.berkeley.edu)) worked on joining station data to IATA data. Worked on EDA for 3 month data along with full datasets and joined data. Helped make presentation and graphics. Worked on L1 regularization.

Thomas Welsh (twelsh@berkeley.edu (mailto:twelsh@berkeley.edu)) worked on final joins and join statistics, deduplication, debugging, time-window engineered variables, and merging code into staging and final notebooks. Wrote content for abstract/submission docs, helped identify EDA tools and troubleshoot modeling and pipeline issues.

Project Report

Abstract

Mitigating the logistical and financial impact of airport delays has never been more important for aviation logistics companies to compete and succeed, and the new forecasting tool from the Flight Delay All-Stars can give these companies 2 hours advance notice if a flight will be delayed by at least 15 minutes, and do so with AUC-ROC performance of over 80.7% (on balanced delay/no-delay samples). The All-Stars accomplished this feat by creating a data pipeline of flight, weather, and airport data, which when processed, feeds a logistic regression model with a log-loss cost function. Blocked time-series cross validated data, rebalanced by undersampling non-delayed flights from 2015-2020, is used to train the model, and 2021 data is used for testing. While features used for the model were selected from the original data pipeline via linear regression using L1 regularization, additional features were engineered, such as the frequency of delays from the same airport in earlier time-windows, the pagerank of the airport graph and an airline ranking. In the final logistic regression model, grid search was used to tune the regularization parameter. Finally, the results were compared to an XGBoost ensemble model.

	Baseline	Logistic	XGBoost
Validation Data AUC	0.5	0.81	0.761
Testing Data AUC	0.5	0.8065	0.7597

Primary Datasets

Data for the model was drawn primarily from 3 datasets: an airline performance dataset and airport dataset from the U.S. Department of Transportation, and a weather dataset from NOAA. These were supplemented by a fourth dataset from the International Air Transport Association (IATA) which published a list of corresponding 3-character airport codes and 4-character airport codes.

More specifically, the Airline On-Time Performance Data comes from the TranStats data collection from the U.S. Department of Transportation, and covers “US certified air carriers that account for at least 1% of domestic scheduled passenger revenues” from 2015 through 2021, comprising 74,177,433 rows x 109 columns in dimension. This data covers several categories of information: datetime information for scheduled and actual departure, arrival, and other activities such as wheels on and wheels off the ground; delays and type of delays; carrier and plane information; and origin and destination information. The import of the data can be seen in Step 1 below. Preliminary EDA (see notebook https://adb-731998097721284.4.azuredatabricks.net/?o=731998097721284#notebook/4236430293322986 (https://adb-731998097721284.4.azuredatabricks.net/?o=731998097721284#notebook/4236430293322986)) as well as Step 2 below shows that there are a mix of numeric and categorical features. As a whole, the dataset is fairly complete, although many data fields are populated by NaN because they track relatively rare events, like the delay caused by a mechanical error, or the time spent on the ground at a diverted airport. However, this appears to be by design and not by incomplete or missing information.

As a whole, there are some noteworthy characteristics of the dataset. First, there are a large number of fields for describing the performance of flights that are diverted in sequence to a series of other airports, but these are rare occurrences, and were ultimately dropped from our final model. Although many features in the flights dataset measure aspects of time, most numeric time-related delays are not normal—many appear to be Poisson distributions. It is also important to note that time is recorded in local time, which means that it will need to be converted before joining it with data that is recorded in UTC standard times.

Weather Data comes from the National Oceanic and Atmospheric Administration repository's Integrated Surface Data (ISD) from 2015 through 2021, and has 898,983,399 rows x 124 columns. The NOAA dataset is very complex, has many missing or incomplete fields. The dataset contains several categories of information: date-time information; weather station location information; the source of the information (often METAR or another source that automatically generates observations at a fixed interval); scientific weather measurements and weather descriptions (wind, pressure, temperature); sky and visibility descriptions; precipitation measurements and descriptions; and other status fields for the sensory equipment.

Although the initial EDA of the weather data revealed that most weather-related fields are actually NaN, the REM (the "remarks") field was consistently populated with METAR data, a coded weather report from the observation station. Parsing this field enables a more complete characterization of the weather information, even if NaNs are common in other fields of the weather dataset. One notable complication to the weather dataset is that while the data is collected at regular intervals, the timestamps do not match up exactly with the flight departure times. To account for this, first the UTC timestamps were stripped from the weather data, and adjusted to local time using the supplementary IATA dataset. Then, a two-hour prior-to-departure time was added to the airlines dataset, and weather data was joined such that the row with the four-character airport code that was closest to the two-hour-prior-to-scheduled-departure was matched to the airline dataset. This last join took our Spark cluster 22 minutes to complete.

Data Pre-Processing

To prepare the data for our model, a number of pre-processing steps were required. First, several engineered variables were added with the intent of improving our performance metrics while at the same time reducing the number of redundant features. (See Step 4 below). One category of variables was time-windowing, essentially looking to see what percentage of flights leaving the same airport (and separately, going to the same destination) in the previous 4 hours that were delayed.

Initially, we experimented with 12-hour look back windows, but found that 4-hour windows improved AUC scores by approximately 5% at that particular time of model analysis.

In addition, we created a PageRank feature to determine which airports are the most associated with delays. Initially, we had created the PageRank score that represented all years of the data. However, we realized that this would lead to data leakage, and avoiding it was a complex endeavor. As a solution, we divided our dataset for a blocked time-series analysis. Of the training set of 2015-2020, we created 6 folds. In each fold, only the delayed January through November flights in each year were counted, and these were used as initial weights for the airport nodes in our graph. We revised our approach to run PageRank for each year between 2015-2020 (excluding December data for each respective year). Subsequently, we took these computed PageRank scores and averaged them to create the PageRank score for 2021. Lastly, we joined the PageRank score back to the main table to have a score for each year and airport.

We also created a feature that ranks airlines by delays. We took the original flights data (not the post-joined data) to ensure that we have all of the flights data and nothing is dropped. Then we filtered to identify flights that were delayed but the delay was partly attributable to the carrier, e.g. CARRIER_DELAY is greater than 0. Rather that taking the number of delays by flights, which gives a skewed view, we calculated for each flight the percentage of delays (flights delayed divided by total flights.). We then ranked the percentage of delays to create our ranking feature. We used a similar treatment for data leakage as PageRank.

At this point, we constructed a model pipeline in Step 5 that would be positioned handle any inputted data. Our model absorbs the raw features provided from Steps 1-4, and then scales the data with a Standard Scaler to ensure that all features are between the ranges of zero and one, and that our categorical variables are one-hot encoded. At this stage, our model had just over 869 features. For each row (data point), Spark then created an 869-point vector, and each of its values were between 0 and 1. (See Step 6.)

In-Pipeline Data Processing

Because fitting a model on 869 features would be very computationally intense, we elected to reduce our variables via linear regression with L1 regularization (see Step 7). This process was more complex than we anticipated because Spark's preference for dealing with vectors rather than dataframes meant that we had to carefully feed the newly created feature vector into the model and then carefully only slice out the indices that we were to keep. Experimentation revealed that an L1 regularization parameter of 0.01 yielded 3 important features. When we first ran L1 regularization, we used a regularization parameter of 0.1 which resulted in no features. Although 3 features may seem small, the three features that are kept are our engineered features 'frac_departures_to_destination_delayed_prev_4_hours', 'frac_origin_departures_delayed_prev_4_hours', 'Airline_Delay_Ranking', and we decided this number of features would be sufficient because they combine aspects of other features.

Our initial EDA indicated that our data was unbalanced--over 80% of the flights had no delays, and less than 20% of the model had delays. This imbalance hinders our ability to train our model for the delay scenario, and consequently we attempted to rebalance our training data before we trained our model. There are two general approaches to solve this problem--oversample the delay examples (SMOTE), or undersample the no-delay examples. Given that fitting our model was already computationally challenging for our cluster resources, and the additional challenges created by implementing SMOTE, we opted for the undersampling approach.

Splitting the time-series data in order to fit our logistic regression model required great care to avoid data leakage. Although 2021 data continued to be held out for final testing, the training data in 2015 through 2020 was split with a blocked time-series cross-validation approach. Each year was applied to a separate fold (6 folds in total), and January through November data was treated as the training data for the fold, and was tested against the December data for that year.

Data Leakage Discussion

The definition of data leakage is when information from outside the training dataset

is used to create the model, and this problem requires careful scrutiny when working with time-series data. In time series analysis, this can mean making a prediction based on data that is from the future at the time of prediction.

The first step we took in ensuring our pipeline has no data leakage was creating a time-series based cross validation strategy. For non-time-series, it is customary to randomly assign data to either the training or validation sets. However, this process would not work for time-series data because it would allow data for training to be used for prediction of the validataion set even though it occurs after the validation data. To solve this problem, we decided to implement a blocked cross-validation method.

Time-series analysis was a new activity for the members of our group and initially we made some mistakes with our engineered features. The first mistake we made was calculating PageRank for the full 6-year training period. To fix this, we calculated PageRank for each fold in our cross validation split to ensure no leakage. Ensuring that there was no data leakage was difficult, and we realized that even basic statistics about our dataset, such as the delay/no-delay ratio that we used for rebalancing, actually required calculation in each fold of our blocked time-series cross validation to ensure that the 2020 delay/no-delay sampling ratio did not

Baseline Model, Logistic Regression Model, and XGBoost Model

Because the ratio of no-delay to delay across the entire dataset was approximately 4:1, by simply predicting the that all flights would be in the no-delay category, we would correctly classify flights the majority of the time. This "baseline" model, while not sophisticated, yields an AUC of approximately 0.50 and serves as a baseline against which we can measure improvements with better models.

We ran two different types of more advanced models in order to predict flight delays. The first was a logistic regression model, which is a relatively simple model that is very straightforward to train and handles binary classification challenges relatively well. However, our primary hyperparameter tuning option was through the regularization parameter, which limited the performance we could produce. We also

experimented with tuning the ElasticNet parameter of the regularization, which uses a hybrid of Lasso and Ridge regression, in order optimize results. However, this experimentation did not lead to any significant changes to the results.

Additionally, we also ran a more sophisticated XGBoost model, which has shown very good results in Kaggle competitions which require binary classification. The XGBoost algorithm improves upon the already powerful tree ensemble by incorporating gradient descent architecture to boost weak learners. XGBoost works great with spark as it can take advantage of distributed computing. Based on several experiments, although the AUC with the more sophisticated XGBoost model did not increase beyond the 0.75-0.76 range, running it on a much larger feature provided very similar results, suggesting that our L1 regression was successful at correctly selecting features while making our algorithms run much faster by reducing unneeded features.

Our trivial baseline model will have no loss function or regularization as it is simply always predicting "no delay". However, we will use a loss function for the two other models that we expect to build. For logistic regression, the team plans to use Log Loss as this is the standard practice when running logistic classification problems:

$$LogLoss = \frac{-1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} * log(p_{ij})$$

where N is the number of samples, M is the number of features, indicates whether ith sample belongs to jth class or not and p indicates the probability of the ith sample belonging to the jth class. We also used the elastic net regularization, a hybrid method which aims to not only minimize the amount of strongly correlated variables but also to make the loss function strongly convex:

$$ElasticNetReg = (1 - \alpha)|\omega_1| + \alpha|\omega_2^2|$$

where is the regularization term (hyper parameter) and w is the sum of the absolute value of the magnitude of the coefficients.

For our tree ensemble model, we had a bit more flexibility in choosing our loss function, and we used the logistic regression loss function and the traditional regularization term used in most XGBoost models:

$$XGBoostRegularizationTerm = \gamma T + \frac{1}{2}\sum_{j=1}^{T}\omega_j^2$$

where w is the vector of leaf scores, T is the number of leaves, and second term is the regularization term.

Results and Gap Analysis

Our models were evaluated with the Area Under the Curve - Receiver Operating Characteristics (AUC-ROC) metric because it works well with classification algorithms, especially with unbalanced outcome variables such as ours. On the 6-fold cross validation, the AUC of the baseline model was 0.50, the AUC of the tuned logistic regression model as 0.82, and the AUC of the tuned XGBoost model was 0.761 for our best run. On the final test set, the AUC of the baseline model was 0.50, the AUC of the tuned logistic regression model as 0.81, and the AUC of the tuned XGBoost model was 0.76. This demonstrates that our models have solid predictive power even though our model outputs seemed relatively insensitive to hyperparameter tuning. Additionally, our L1 regularization only kept 3 features which means this may have been excessively aggressive. We would pursue additional exploration of this hyperparameter with additional time resources at our disposal.

It is important to note that in our cross-validation studies, the AUC was the average composite of all 6 folds. However, we noticed that the last year of the study 2020, scored consistently higher than the other folds, no matter which model we used. Our intuition is that the 2020 was a year in which travel was greatly impacted by the pandemic, and with fewer travelers, airlines had extra capacity and thus fewer cascading delays. Because cascading delays involve a chain of events and are harder to predict, our models may have performed better when there were fewer of them and had more direct, generic causes.

For our gap analysis, we compared our AUC metric to all other groups that reported this metric at the time of writing this analysis (11:16pm Eastern Time 8/4). Groups reported AUC scores of 0.61, 0.68, 0.65, and 0.60. This means that our best score of 0.81 puts us at the top of all groups that decided to use AUC as their metric. The next closest is 0.13 AUC score away. This validates that our developed featuers were highly predictive.

Learnings and Discussion

With the benefit of hindsight, if we had to do the project again, there are many areas where we would take a different approach. Although avoiding the technical mistakes that created difficulties early on would certainly have been helpful, improvements to our model-building process likely would have resulted in even better outcomes.

First, although we performed what we thought was a thorough EDA in the early phases of our project, only in later phases did we fully appreciate the importance of doing EDA at every step. For instance, in addition to the data we received from databases, we should have run an EDA on each new feature that we engineered. By stress-testing each of our engineered features as thoroughly as we vetted the input features, we would have had better data with which to compare different versions of each engineered feature. Also, we after Phase 1 and 2, which required substantial EDA, we elected to use the class-provided joined dataset instead of our own, because we thought that it would save us time because the METAR weather variables were already parsed. However, many of the features in the provided joined dataframe were different than the ones we created in Phases 1 and 2, and we should have rerun an EDA on the new joined dataframe. Several pre-processing functions that we created to work on our original version of the dataframe did not work as planned on the provided one, and that took us a lot more time than we expected to debug.

After Phases 1 and 2, we were very excited to have a complete, joined dataset that we saved to the blob, but we were mistaken in thinking that writing something to the blob signifies that we had completed something. Rather than using this as a starting point or floor for our subsequent work, we would have been much better off if we

had gone back to the smaller, 3-month dataset that we originally used, and perfected building our data processing pipeline and model on this smaller dataset. Had we done that, we could have iterated much faster and performed more experiments, rather than waiting for our cluster to churn through large datasets.

Part of the challenge was that we tried to include as many features as possible in the pipeline in order to be thorough, with the expectation that a linear regression with L1 regularization would help us winnow down our features to just the key features. There were several problems with this let-the-model-figure-it-out approach. First, there were so many features that we didn't scrutinize each one carefully enough--our pipeline mistakenly classified many binary encoded features as categorical, and therefore exploded them further into unnecessary separate features. Second, just because would could create one-hot-encoded variables from categorical features like airport_codes and tail_numbers does not mean that we should. We could have tested to see if a subset of these features was useful before devoting considerable computational resources to find needles in haystacks. Had we thought more clearly, rather than trying to one-hot encode tail numbers, we could have been more creative about how we incorporated the tail number information into our model, like how perhaps may have been better utilized as part of an engineered graph-like feature rather than a one-hot encoded one.

Because we had a lot of training in L1 regularization and felt comfortable with using it, we did not spend enough time exploring if there were better methods for systematic feature selection in a dataset this large. Consequently, we did not spend enough time looking at many other techniques which may have been more efficient. Since feature selection was a significant part of our computational expenditure, we would have been well served by finding a more efficient process. Many features that we anticipated would be significant, like pagerank, did not survive the L1 regularization, and we would have liked to have explored why if we had additional time and resources. Furthermore, we would have liked to experiment further with many more folds that our current 6-fold process.

For the members of our group, this was the largest data science project that we had tackled as a team. If we had more time, we definitely would have invested more tools to measure our performance and collaborate more effectively. With each team member working in his own version of the notebook, on a different parts of the

pipeline, on different subsets of features, it was difficult to run experiments that generated apples-to-apples results. In retrospect, it may been much more effective to get an end-to-end pipeline running first on a small dataset with a small number of features, and then backfill pipeline activites like feature selection on larger datasets.

While this was not something that we could control in this particular project, having full control of our cluster could have made many of our tasks easier. For instance, if we had greater ability to adjust the number of workers in our cluster, and tune other settings, we think we may have been able to get some of our tasks to run more effectively.

If we had more time, there are many improvements that we would make to our analysis. One of the most important would be a deeper dive into feature analysis and the relative importance of each of the features that we kept. L1 Regularization selected features that aligned with many of our expectations for what might be important features, but we did not discover a method of ranking the feature importance in our current implementation. Having that data would be useful not only to our audience, but it would provide clues on where to focus future efforts. In a similar vein, we wished that we had done more correlation studies between features. If our goal is to provide means for clients to mitigate the effects of delays, they would need to know how much impact an action on an individual variable would have.

Conclusion

Predicting airport delays is crucial for aviation logistics companies to be competitive. The Flight Delay All-Stars have set out to develop a tool that can give these companies 2 hours notice if a flight will be delayed by 15 or more minutes. We hypothesized that a machine learning pipeline with unique features can accurately predict flight delays and we have proven this with our pipeline showcased above. Some of the interesting aspects about our pipeline include the engineering of novel features like airport pagerank and time window analysis, feature selection using L1 regularization, and blocked time-series cross validation. This resulted in an AUC of

81%. For our modeling, we compared a logistic regression model, and an XGBoost model to our baseline and we improved our baseline by 21% AUC. This put us as the top AUC score at the time of checking.

Step 1a: Import Libraries

```
from pyspark.sql.functions import col, when, substring
from pyspark.sql.functions import *
import pyspark.sql.functions as F
from pyspark.sql.types import StringType, IntegerType, BinaryType,
UserDefinedType, FloatType, StructType, StructField


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# import seaborn as sns
import datetime as dt
import itertools
import re

from pyspark.ml.linalg import DenseVector, SparseVector, Vectors
from pyspark.ml.feature import VectorAssembler, StandardScaler, Imputer,
StringIndexer, OneHotEncoder, VectorSlicer

from pyspark.ml import Pipeline
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml.classification import RandomForestClassifier,
DecisionTreeClassifier, LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator,
BinaryClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics,
MulticlassMetrics
from pyspark.ml.regression import LinearRegression, LinearRegressionModel
from sparkdl.xgboost import XgboostClassifier

from hyperopt import STATUS_OK, Trials, fmin, hp, tpe

!pip install -U airportsdata
import airportsdata
from graphframes import *
print("Welcome to the W261 final project - Team 18 in the house!")
from pyspark.sql.functions import monotonically_increasing_id,row_number
from pyspark.sql.window import Window

spark.sparkContext.addPyFile("dbfs:/custom_cv.py")
from custom_cv import CustomCrossValidator

Requirement already satisfied: airportsdata in /databricks/python3/lib/python3.
8/site-packages (20220731)
WARNING: You are using pip version 21.0.1; however, version 22.2.2 is availabl
```

```
e.
You should consider upgrading via the '/databricks/python3/bin/python -m pip in
stall --upgrade pip' command.
Welcome to the W261 final project - Team 18 in the house!
```

```
# Inspect the Mount's Final Project folder
# Please IGNORE dbutils.fs.cp("/mnt/mids-
w261/datasets_final_project/stations_data/", "/mnt/mids-
w261/datasets_final_project_2022/stations_data/", recurse=True)
data_BASE_DIR = "dbfs:/mnt/mids-w261/datasets_final_project_2022/"
```

```
# Setting up blob container to minimize repetitive calculation
blob_container = "w261blobalex" # The name of your container created in
https://portal.azure.com
storage_account = "w261alex" # The name of your Storage account created in
https://portal.azure.com
secret_scope = "first_scope" # The name of the scope created in your local
computer using the Databricks CLI
secret_key = "first_key" # The name of the secret key created in your local
computer using the Databricks CLI
blob_url = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"
mount_path = "/mnt/mids-w261"
blob_container = "w261blobalex"
storage_account = "w261alex"
blob_url = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"

spark.conf.set(
  f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
  dbutils.secrets.get(scope = secret_scope, key = secret_key)
)
```

```
#helper functions to save to the blob
def save_to_blob(obj, path_str):
  """
  Saves the obj to the blob at the provided file name
  """

  path = f"{blob_url}/{path_str}"

  try:
    obj.write.format("delta").mode("overwrite").save(path)
  except:
    print('This File already exists')

def load_blob_obj(path_str):
  """
  Loads object to the notebook that is saved into the blob
  """
  path = f"{blob_url}/{path_str}"

  try:
    return spark.read.format("delta").load(path)
  except:
    print(f"no file name {path_str} exists")
```

Step 1b: Import Data

```
# # Load Full (Provided) DataFrames
# df_airlines = spark.read.parquet(f"{data_BASE_DIR}parquet_airlines_data/")
# df_weather = spark.read.parquet(f"{data_BASE_DIR}parquet_weather_data/")
# df_stations = spark.read.parquet(f"{data_BASE_DIR}stations_data/*")
```

```
# # Load IATA Dataframe (NOT PROVIDED) for the translating 4-character Airport
Call data to 3-character IATA airport codes (very small dataset) and adding
timezone information to airports
# airports = airportsdata.load()
# iata = pd.DataFrame.from_dict(airports).transpose()
# iata.drop(['lat','lon'], axis = 1, inplace=True)
# iata = iata[iata['country'] == "US"]
# iata = iata[iata['iata'] != ""]
# iata.drop(['city','subd','country','elevation'], axis = 1, inplace = True)
# iata.rename({'icao':'neighbor_call'}, axis = 1, inplace = True)
# df_iata = spark.createDataFrame(iata)
# df_iata.distinct()
# print("IATA DATAFRAME:")
# df_iata.display()
```

Step 2: Exploratory Data Analysis

Note: EDA was primarily conducted in this separate notebook ($./w261_Final_Project_Team_18_Phase_1_and_2_Final). Key parts of that EDA (but not all of it) are included below as commented code for reference.

```
# # Run on Full airline Data
# dbutils.data.summarize(df_airlines)
```

Per the table above, the raw airline data has 74.2M records. There are 18 numerical features in the airline data that have over 80% null values. These features are dropped. We can see from our target variable "DEP_DEL15" that the mean is 0.18. That means that 18% of our flights are delayed and the rest of the flights are not delayed. Also interesting is the "DEP_DELAY" feature which has a mean of 9.35. This says that the mean delay time is just over 9 minutes for all flights in this dataset. Moving on to the categorical variables, an interesting tidbit is that there are 379 unique "ORIGIN" airports but only 377 unique "DEST" airports. This means that there are 2 airports that are only used for departing flights but no arriving flights.

Similar to the numerical features, there are several features that have a lot of Null values. There are 36 columns that have greater than 90% nulls from the categorical variables in the airline data and these features will be dropped

```
# # Run on Full Weather Data
# dbutils.data.summarize(df_weather)
```

Per the table above, we can see the raw weather dataset has 899 million rows. Most of the features in this dataset have almost all Null values which we can see when examining the "missing" column above. At face value it appears that this data is mostly unusable, but we are able to impute many of the values by getting data from the FM-15 report. This method will be described further in the data processing pipeline section as we have to extract METAR data to get anything useful from this data due to the large number of Nulls.

```
# # Run on Full Stations Data

# dbutils.data.summarize(df_stations)
```

The stations data has 5 million rows with zero null values in the entire dataset. This data compares two stations to each other and shows the distance between them. One interesting aspect of this dataset is the distribution of the "distance_to_neighbor" field. The distribution is right skewed and this means that most of the stations are within a reasonable distance of each other and there are a few stations that are very far away. One can also see this by examining the mean / median field because the mean is much higher than the median.

Step 3: Data Pre-Processing and Joining

```
# # drop duplicate rows for airlines, select columns
# df_airlines = df_airlines.dropDuplicates(['ORIGIN', 'DEST', 'DEP_TIME',
'CRS_DEP_TIME', 'TAIL_NUM'])\
#                          .select('ORIGIN', 'DEST',
#                                  'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY',
#                                  'DEP_DELAY_NEW', 'DEP_DEL15',
'DEP_DELAY_GROUP',
#                                  'DEP_TIME_BLK', 'CRS_ARR_TIME', 'ARR_TIME',
#                                  'ARR_DELAY', 'ARR_DELAY_NEW', 'ARR_DEL15',
#                                  'ARR_DELAY_GROUP', 'ARR_TIME_BLK',
'CANCELLED',
#                                  'CRS_ELAPSED_TIME', 'ACTUAL_ELAPSED_TIME',
'AIR_TIME', 'TAIL_NUM',
#                                  'FLIGHTS', 'DISTANCE', 'CARRIER_DELAY',
#                                  'WEATHER_DELAY', 'NAS_DELAY',
'SECURITY_DELAY',
#                                  'LATE_AIRCRAFT_DELAY', 'FL_DATE')

# # select just those stations with distance to themselves of 0--these are the
airports
# # choose columns, and keep distinct ones
# df_stations = df_stations.filter(col("distance_to_neighbor") == 0)\
#                 .select(['neighbor_call', 'neighbor_name', 'neighbor_state',
'lat', 'lon'])\
#                 .distinct()

# # Gather information about dataframes just prior to join for later comparison
# df_airlines_priortojoinwithstations_length = df_airlines.count()
# df_stations_priortojoinwithairlines_length = df_stations.count()
# df_airlines_priortojoinwithstations_width = len(df_airlines.columns)
# df_stations_priortojoinwithairlines_width = len(df_stations.columns)

# # Select weather columns for use
# # Select weather data from only North America (LATITUDE in stations went from
18.25 (PR) to 71.333 (AK), and LONGITUDE went from -176.65 (AK) to -66.098 (PR)
# # Note: EARECKSON AIR STATION AIRPORT seems to have a bad lon point of 174.1
instead of -174.1), but not pulled through to df_main
# # Then add timestamp column that will match format with df_airlines later on
# # Drop the duplicate weather reports--by using orderBy descending before
dropDuplicate, we keep the weather report that is closest
# # timestamp_utc of the weather that is closest to 2 hours prior to scheduled
departure

# df_weather = df_weather.select('STATION', 'DATE', 'LATITUDE', 'LONGITUDE',
'ELEVATION',
```

```
#                                 'REPORT_TYPE', 'SOURCE',
'HourlyAltimeterSetting',
#                                 'HourlyDewPointTemperature',
'HourlyDryBulbTemperature',
#                                 'HourlyPrecipitation',
'HourlyPresentWeatherType', 'HourlyRelativeHumidity',
#                                 'HourlySkyConditions',
'HourlyStationPressure', 'HourlyVisibility',
#
'HourlyWetBulbTemperature','HourlyWindDirection','HourlyWindSpeed','REM','YEAR'
)\
#                         .filter((df_weather['LATITUDE'] > 15) &
(df_weather['LATITUDE'] < 75) & (df_weather['LONGITUDE'] < (-70)) &
(df_weather['LONGITUDE'] > (-178)) & (df_weather['REPORT_TYPE'] == 'FM-15'))\
#                         .withColumn('timestamp_date',to_timestamp(col('DATE'),
"yyyy-MM-dd'T'HH:mm:ss"))

# df_weather = df_weather.withColumn("timestamp_utc", F.date_trunc("hour",
df_weather.timestamp_date))\
#                         .orderBy(col('DATE').desc())
```

```
# # Join df_iata to df_stations to get 3-code airport ID and timezone -- you
will need this for joining the 3 main dataframes
# df_stations = df_stations.join(df_iata, df_stations['neighbor_call'] ==
df_iata['neighbor_call'], "left").drop(df_stations['neighbor_call'])
# df_stations = df_stations.select(['iata', 'name', 'neighbor_call',
'neighbor_name', 'neighbor_state', 'lat', 'lon', 'tz'])
# # print("STATIONS DATAFRAME:")
# # df_stations.display()

# # Join df_stations to df_airlines on 3-code airport ID
# df_airlines = df_airlines.join(df_stations, df_airlines['ORIGIN'] ==
df_stations['iata'], "left").drop(df_airlines['ORIGIN'])
# # print("AIRLINES DATAFRAME:")
# # df_airlines.display()

# # Gather post-join statistics
# df_airlines_afterjoinwithstations_length = df_airlines.count()
# df_airlines_aftertojoinwithstations_width = len(df_airlines.columns)

# # Helper functions for extracting airport_call data from METAR field in
df_weather further below
# def clean_metar(obs):
#   return str(re.sub('^\S* ', '', str(obs)))

# def strip_airport(obs):
#   airport_search =  re.search(' (K\S+)', str(obs), re.IGNORECASE)
#   if airport_search:
#     return airport_search.group(1)
#   else:
#     return str(re.sub(' .*$', '', str(obs)))

# newREM = udf(lambda x: clean_metar(x), StringType())
# airport_name = udf(lambda x: strip_airport(x), StringType())

# # METAR and airport_call columns added to the weather database
# df_weather = df_weather.withColumn("METAR", newREM(col('REM')))\
#                        .withColumn('airport_call',
airport_name(col('METAR')))

# # Remove duplicates - latest eligible timestamp kept (because df_weather
ordered by 'DATE' above)
# df_weather = df_weather.dropDuplicates(['timestamp_utc', 'airport_call'])

# # print("WEATHER DATAFRAME:")
# # df_weather.display()
```

```
# # Timestamp processing
# ## Part 1 (Convert to Date Time)
# # Cast Computer Reservation System Departure Time to string
#
df_airlines=df_airlines.withColumn('timestamp',df_airlines.CRS_DEP_TIME.cast(St
ringType()))

# # Add 1 leading zero to times with length 3 to make timestamps consistent
(example: 949 gets updated to 0949, thereby giving each timestamp a length of
4)
# df_airlines = df_airlines.withColumn('timestamp', F.lpad(F.col('timestamp'),
4, '0'))

# # Add 2 trailing zeroes to give timestamp seconds for comparison to weather
data
# df_airlines = df_airlines.withColumn('timestamp', F.rpad(F.col('timestamp'),
6, '0'))

# # Concatenate date and time
#
df_airlines=df_airlines.withColumn('timestamp',concat(df_airlines.FL_DATE,lit('
T'), df_airlines.timestamp))

# # Convert date and time string to timestamp
# df_airlines =
df_airlines.withColumn('timestamp_string',to_timestamp(col('timestamp'), "yyyy-
MM-dd'T'HHmmss"))
# # df_airlines.select('timestamp_string').show()
```

```
# ## Part 2(Update Date Time)

# # Round hour down (to join to weather data because it is rounded)
# df_airlines=df_airlines.withColumn("rounded_timestamp", F.date_trunc("hour",
df_airlines.timestamp_string))

# # Substract 2 hours
# df_airlines =
df_airlines.withColumn("minus_2hours_timestamp",col("rounded_timestamp") -
expr("INTERVAL 2 HOURS"))

# df_airlines.createOrReplaceTempView("airlines") # Create a temporary view
# df_airlines = spark.sql("SELECT *, to_utc_timestamp( minus_2hours_timestamp,
tz) as FINAL_DEP_TIME_tz FROM airlines")

# df_airlines.createOrReplaceTempView("airlines") # Create a temporary view
# spark.sql("SELECT FL_DATE,CRS_DEP_TIME,timestamp,
timestamp_string,rounded_timestamp,minus_2hours_timestamp, FINAL_DEP_TIME_tz,
tz FROM airlines WHERE CRS_DEP_TIME IS NOT NULL").show()



# # Final join of airlines and weather on time and airport

# # Measure size of pre-joined airlines and weather dataframes
# df_airlines_priortojoinwithweather_length = df_airlines.count()
# df_weather_priortojoinwithairlines_length = df_weather.count()
# df_airlines_priortojoinwithweather_width = len(df_airlines.columns)
# df_weather_priortojoinwithairlines_width = len(df_weather.columns)

# # Join occurs here
# df_main = df_airlines.join(df_weather, (df_airlines['neighbor_call'] ==
df_weather['airport_call']) & (df_airlines['FINAL_DEP_TIME_tz'] ==
df_weather['timestamp_utc']), "left")

# # Measure size of final dataframe after join
# df_main_length = df_main.count()
# df_main_width = len(df_main.columns)
# df_main.display()
```

```
# # Special note:
# # df_main above has null values for the following airports:
# # HNL Honolulu International Airport PHNL
# # SIT Sitka Rocky Gutierrez Airport PASI
# # OGG Kahului Airport PHOG
# # When you look at the weather dataset below, there are only 7 rows of
weather data for those locations.
# # We will explore how best to handle these errors at a later date

# filter_values_list =['PNHL', 'PASI', 'PHOG']
# df_test = df_weather.filter(df_weather.airport_call.isin(filter_values_list))
# df_test.display()



# # Join Statistics:
# print('Join Statistics')
# print('-------------------------------------------------')

# print('\nInitial Dataframe Sizes:')
# print(f'Airlines data size: {df_airlines_initial_length} rows x
{df_airlines_initial_width} columns')
# print(f'Weather data size: {df_weather_initial_length} rows x
{df_weather_initial_width} columns')
# print(f'Stations data size: {df_stations_initial_length} rows x
{df_stations_initial_width} columns')

# print('\nAirlines-Stations Join:')
# print(f'Airlines size prior to join:
{df_airlines_priortojoinwithstations_length} rows x
{df_airlines_priortojoinwithstations_width} columns')
# print(f'Stations size prior to join:
{df_stations_priortojoinwithairlines_length} rows x
{df_stations_priortojoinwithairlines_width} columns')
# print(f'Airlines size after join: {df_airlines_afterjoinwithstations_length}
rows x {df_airlines_aftertojoinwithstations_width} columns')

# print('\nAirlines-Weather Join:')
# print(f'Airlines size prior to join:
{df_airlines_priortojoinwithweather_length} rows x
{df_airlines_priortojoinwithweather_width} columns')
# print(f'Weather size prior to join:
{df_weather_priortojoinwithairlines_length} rows x
{df_weather_priortojoinwithairlines_width} columns')
# print(f'Main dataframe size after join: {df_main_length} rows x
{df_main_width} columns')
```

```
# # Save by Writing Joined Dataset to Blob
# blob_container = "w261blobalex"
# storage_account = "w261alex"
# blob_url =
f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"

# # Command to write to team blob (Assume we have some df called df_main)
# df_main.write.parquet(f"{blob_url}/df_main_005")
```

Special Note: to accellerate completion and debugging of the data cleansing and joining process, Team18 elected to use the provided cleaned, joined dataset.

```
# Read in Provided Joined Dataset from Blob (stored in Delta Lake format)
DELTALAKE_GOLD_PATH = f"{blob_url}/df_lite_deltalake011"
# Re-read as Delta Lake
df_lake = load_blob_obj('df_lite_deltalake011')

# Review data
display(df_lake)
```

	_utc_dept_ts	_utc_dept_minus2_ts	_utc_de
1	2018-03-15T17:15:00.000+0000	2018-03-15T15:15:00.000+0000	2018-03
2	2018-01-22T02:20:00.000+0000	2018-01-22T00:20:00.000+0000	2018-01-
3	2018-08-19T12:00:00.000+0000	2018-08-19T10:00:00.000+0000	2018-08
4	2018-08-26T18:25:00.000+0000	2018-08-26T16:25:00.000+0000	2018-08
5	2018-03-15T17:59:00.000+0000	2018-03-15T15:59:00.000+0000	2018-03
6	2018-02-15T15:04:00.000+0000	2018-02-15T13:04:00.000+0000	2018-02
7	2018-08-19T12:20:00.000+0000	2018-08-19T10:20:00.000+0000	2018-08

Truncated results, showing first 1000 rows.

Step 4: Feature Engineering

In addition to the features that were included in the airlines, weather, and stations datasets, we engineered a few additional features. The first two were time-window features which showed, for a given flight, the percentage of flights that were delayed leaving from the same origin airport in the previous 12 hours, and the percentage of flights that were delayed leaving from the same destination airport in the previous 12-hours. This feature seemed potentially useful because when other flights leaving an airport are delayed, it is more likely that yours will be also. Additionally, we filtered all of the flights that were delayed in our training set, and implemented from the directional graph of airport flights, which airports had the highest pagerank in terms of delays. We interpreted this as certain airports were more important for delays across the entire airline network. Reviewing the data, we can see that it aligns with our expectations: ORD, ATL, and DFW are major airport hubs, and delays there can cascade through the network.

```
# Add additional columns for time
df_lake = df_lake.withColumn('DEP_TIME', F.lpad(F.col('DEP_TIME'), 4, '0'))

#Create hour and min features to be used as categorical features later
#create "minutes from passed in the day" variable with udf below and use as
numerical feature
def time_to_min(hour, minute):
  return (hour*60) + minute

to_min = udf(time_to_min, IntegerType())


df_lake = df_lake.withColumn('hour', col('DEP_TIME').substr(1,2))\
                 .withColumn('min', col('DEP_TIME').substr(3,2))\
                 .withColumn('min_of_day', time_to_min(col('hour'),
col('min')))


#removing all rows with missing outcome variable
df_lake = df_lake.na.drop(subset = ['DEP_DEL15'])
```

Time Window Analysis: examine flight delays to/from the same airport over windows just before the scheduled departure. Our EDA informed us that this could be an important feature because when looking at percentage of delays by airport, we

noticed a huge spread with the worst airports having delays over 40% and the best airports only had around 1% so taking a snapshot of the airport at the time of prediction we think will be a good feature.

```
# Time Window Analysis

# Here we examine how many other flights from the same origin were delayed in
the last 4 hrs
# and how many flights to the same destination were delayed over the past 4
hrs.

# Add a timestamp column (in correct format) for windowing
df_lake = df_lake.withColumn('_utc_dept_ts_minus2_timestamp_version',
concat(col('_utc_dept_minus2_ts').substr(1,10),lit('
'),col('_utc_dept_minus2_ts').substr(12,19)).cast('timestamp').cast('long'))

# Create windows for looking at X numbers of hours in the past
hours_window = lambda i: i * 3600       # 3600 seconds in an hour
w0 = Window.partitionBy(F.col('ORIGIN'))  # To examine flights from the same
origin
w1 = Window.partitionBy(F.col('DEST'))    # To examine flights to the same
destination

# For a given origin, create columns of departures in last 4 hrs,
#                                       delayed departures in last 4 hrs,
#                                       fraction of delayed/total departures in
4 hrs
df_lake = df_lake.withColumn('scheduled_origin_departures_prev_4_hours',
F.count('*').over(w0.orderBy(F.col('_utc_dept_ts_minus2_timestamp_version')).ra
ngeBetween(-hours_window(4),0)))\
    .withColumn('origin_departures_with_delay_prev_4_hours',
F.sum('DEP_DEL15').over(w0.orderBy(F.col('_utc_dept_ts_minus2_timestamp_version
')).rangeBetween(-hours_window(4),0)))\
    .withColumn('frac_origin_departures_delayed_prev_4_hours',
(F.col('origin_departures_with_delay_prev_4_hours')/F.col('scheduled_origin_dep
artures_prev_4_hours')))


# For a flights to a given destination, create columns of departures in last 4
hrs,
#                                                       delayed departures in
last 4 hrs,
#                                                       fraction of
delayed/total departures in 4 hrs
df_lake =
df_lake.withColumn('scheduled_departures_to_destination_prev_4_hours',
F.count('*').over(w1.orderBy(F.col('_utc_dept_ts_minus2_timestamp_version')).ra
ngeBetween(-hours_window(4),0)))\
```

```
    .withColumn('departures_to_destination_with_delay_prev_4_hours',
F.sum('DEP_DEL15').over(w1.orderBy(F.col('_utc_dept_ts_minus2_timestamp_version
')).rangeBetween(-hours_window(4),0)))\
    .withColumn('frac_departures_to_destination_delayed_prev_4_hours',
(F.col('departures_to_destination_with_delay_prev_4_hours')/F.col('scheduled_de
partures_to_destination_prev_4_hours')))\
    .drop('_utc_dept_ts_minus2_timestamp_version')
```

Graph Feature Analysis: PageRank

```
# Rather than applying pageRank to determine most popular airports, we apply
PageRank to delayed flights to determine which airport is the most associated
with delays.

# Create empty dataframe with specified schema
schema = StructType([
  StructField('id', StringType(), True),
  StructField('pagerank', FloatType(), True),
  StructField('pagerank_year', StringType(), True)
  ])

pagerank_all = spark.createDataFrame([], schema)
pagerank_all.printSchema()
```

```
root
 |-- id: string (nullable = true)
 |-- pagerank: float (nullable = true)
 |-- pagerank_year: string (nullable = true)
```

```
# 2 phased approach: 1) Calculate pageRank for each year (2015, 2016, 2017,
2018, 2019, and 2020) using Jan-November data for each of
# these respective years 2) Take average by year for each carrier to arrive at
the 2021 pagerank numbers
years=[2015,2016,2017,2018,2019,2020] # selected years
for selected_year in years:
  df_delayed_flights = df_lake.where((df_lake.DEP_DEL15=='1') & (df_lake.YEAR
== selected_year) & (df_lake.MONTH!=12))

  # Get nodes by using airport origin and destination then unioning them
distinctly
  nodes_origin = df_delayed_flights.select(df_delayed_flights["ORIGIN"])
  nodes_dest = df_delayed_flights.select(df_delayed_flights["DEST"])

  nodes = nodes_origin.union(nodes_dest).distinct()
  nodes = nodes.withColumnRenamed("ORIGIN","id")
  #nodes.display()

  airport_vertices = (
      nodes
      .select('id')
  )

  airport_edges = (
      df_delayed_flights
      .select(
          F.col('ORIGIN').alias('src'),
          F.col('DEST').alias('dst')
      )
  ).cache()

  # create GraphFrame
  graph = GraphFrame(airport_vertices, airport_edges)


  # run pageRank for each year
  results = graph.pageRank(resetProbability=0.15, maxIter=10)

  pagerank_delays = results.vertices.select("id", "pagerank")
  pagerank_delays = pagerank_delays.withColumn("pagerank_year",
lit(selected_year)) #add respective year
  pagerank_all = pagerank_all.union(pagerank_delays) # append pagerank values
by year to main pagerank table

# perform pagerank calculations for year 2021 and add back to main pagerank
table
```

```
pagerank_2021 = pagerank_all.groupBy('id').avg('pagerank')
pagerank_2021 = pagerank_2021.withColumnRenamed('avg(pagerank)', 'pagerank')
pagerank_2021 = pagerank_2021.withColumn("pagerank_year", lit(2021))
pagerank_all = pagerank_all.union(pagerank_2021)



# join pagerank info back to the df_lake table
pagerank_all  = pagerank_all.withColumn("pagerank_year",
pagerank_all["pagerank_year"].cast("int"))

df_lake.createOrReplaceTempView("lake")
pagerank_all.createOrReplaceTempView("pagerank_all")

df_lake = spark.sql("SELECT lake.*, pagerank_all.pagerank FROM lake LEFT JOIN
pagerank_all ON lake.YEAR = pagerank_all.pagerank_year AND lake.ORIGIN =
pagerank_all.id")



###Airlines Ranking by Percent of Delays
# use airlines data to calculate percent of delays so nothing has been filtered
by joins

df_airlines = spark.read.parquet(f"{data_BASE_DIR}parquet_airlines_data/")
df_airlines.createOrReplaceTempView("airlines")

df_total_flights= spark.sql("SELECT YEAR, OP_UNIQUE_CARRIER,
COUNT(OP_UNIQUE_CARRIER) AS Total_Flights FROM airlines WHERE MONTH!=12 AND
YEAR!=2021  GROUP BY YEAR, OP_UNIQUE_CARRIER  ") # total flights

df_total_flights.createOrReplaceTempView("total_flights") # Create a temporary
view

df_delayed_airlines = df_airlines.where((df_airlines.CARRIER_DELAY>0)&
(df_airlines.YEAR != 2021) & (df_airlines.MONTH!=12)) # get delayed flights
where delay attributed to carrier

df_delayed_airlines=df_delayed_airlines.withColumn("YEAR", year("FL_DATE")) #
get year of flights

df_delayed_airlines.createOrReplaceTempView("delayed_airlines") # Create a
temporary view
```

```
df_airlines_delays = spark.sql("SELECT OP_UNIQUE_CARRIER, YEAR,
COUNT(OP_UNIQUE_CARRIER) AS Carrier_Delays FROM delayed_airlines GROUP BY
OP_UNIQUE_CARRIER, YEAR ORDER BY Carrier_Delays desc") # delayed flights by
carrier and year

df_airlines_delays.createOrReplaceTempView("airlines_delays")


# join total flights data to delayed flights by carrier data
# calculate percentage of delays (delays/total flights)

df_airlines_rank = spark.sql("SELECT airlines_delays.OP_UNIQUE_CARRIER,
airlines_delays.YEAR, airlines_delays.Carrier_Delays,
total_flights.Total_Flights,
(airlines_delays.Carrier_Delays/total_flights.Total_Flights) AS
Percent_of_Delays FROM airlines_delays LEFT JOIN total_flights ON
airlines_delays.YEAR = total_flights.YEAR AND airlines_delays.OP_UNIQUE_CARRIER
= total_flights.OP_UNIQUE_CARRIER")


# calculate percent of delays for year 2021 by taking average of previous years

df_airlines_rank_2021 =
df_airlines_rank.groupBy('OP_UNIQUE_CARRIER').avg('Percent_of_Delays')
df_airlines_rank_2021 =
df_airlines_rank_2021.withColumnRenamed('avg(Percent_of_Delays)',
'Percent_of_Delays')
df_airlines_rank_2021 = df_airlines_rank_2021.withColumn("YEAR", lit(2021))
df_airlines_rank =
df_airlines_rank.select("OP_UNIQUE_CARRIER","Percent_of_Delays","YEAR")
df_airlines_rank = df_airlines_rank.union(df_airlines_rank_2021)


# rank by percentage of delays

windowSpec  = Window.partitionBy("YEAR").orderBy("Percent_of_Delays")

df_airlines_rank =
df_airlines_rank.withColumn("Airline_Delay_Ranking",row_number().over(windowSpe
c))
df_airlines_rank.display()
```

	OP_UNIQUE_CARRIER	Percent_of_Delays	YEAR	Airline_Delay_R
1	AS	0.04816464425352077	2015	1
2				

3	VX	0.06922244496088276	2015	3
4	DL	0.07104820662608259	2015	4
5	AA	0.09332144583953116	2015	5
6	EV	0.09451592063458984	2015	6
7	MQ	0.09452965403323357	2015	7

Showing all 109 rows.

```
df_lake.createOrReplaceTempView("lake")
df_airlines_rank.createOrReplaceTempView("airlines_rank")

df_lake = spark.sql("SELECT lake.*, airlines_rank.Airline_Delay_Ranking FROM
lake LEFT JOIN airlines_rank ON lake.YEAR = airlines_rank.YEAR AND
lake.OP_UNIQUE_CARRIER = airlines_rank.OP_UNIQUE_CARRIER")



# Sanity check (does pagerank scores intuitively align to delayed flights
"from" and "to" airports)

df_delayed_flights.createOrReplaceTempView("delayed_flights") # Create a
temporary view

spark.sql("SELECT ORIGIN, COUNT(ORIGIN) AS Delays FROM delayed_flights GROUP BY
ORIGIN ORDER BY Delays desc").display()

spark.sql("SELECT DEST, COUNT(DEST) AS Delays FROM delayed_flights GROUP BY
DEST ORDER BY Delays desc").display()
```

	ORIGIN	Delays
1	DFW	23493
2	ATL	18497
3	ORD	16246
4	DEN	15868
5	CLT	15262
6	LAX	9076
7	PHX	8771

Showing all 358 rows.

	DEST	Delays
1	DFW	19697
2	ATL	16391
3	ORD	14815

4	CLT	13847
5	DEN	13436
6	LAX	9554
7	LAS	8697

Showing all 360 rows.

```
df_lake.printSchema()
```

```
root
 |-- _utc_dept_ts: timestamp (nullable = true)
 |-- _utc_dept_minus2_ts: timestamp (nullable = true)
 |-- _utc_dept_actual_ts: timestamp (nullable = true)
 |-- _utc_arr_ts: timestamp (nullable = true)
 |-- _utc_arr_actual_ts: timestamp (nullable = true)
 |-- QUARTER: integer (nullable = true)
 |-- MONTH: integer (nullable = true)
 |-- DAY_OF_MONTH: integer (nullable = true)
 |-- DAY_OF_WEEK: integer (nullable = true)
 |-- FL_DATE: string (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- ORIGIN_STATE_FIPS: integer (nullable = true)
 |-- DEST: string (nullable = true)
 |-- DEST_STATE_FIPS: integer (nullable = true)
 |-- DEP_TIME: string (nullable = true)
 |-- DEP_DELAY_NEW: double (nullable = true)
 |-- DEP_DEL15: double (nullable = true)
 |-- DEP_DELAY_GROUP: integer (nullable = true)
 |-- DEP_TIME_BLK: string (nullable = true)
 |-- ARR_DELAY_NEW: double (nullable = true)
```

Step 5: Model Pipeline

Below we create our functions to set up our pipeline. This includes setting up methods for one hot encoding for categorical variables and imputing the median value for numerical variables. Based on EDA, we decided that only 58 categorical features and 32 numeric features were sufficiently complete would have potential for predictive power and therefore potential inculsion in our model.

```
# Categorical features need to be handled differently than numeric ones


# Revised TW set
categoricals =
['QUARTER','MONTH','DAY_OF_MONTH','DAY_OF_WEEK','ORIGIN','DEST','origin_airport
_type','dest_airport_type','YEAR']

numerics =
['DISTANCE','origin_weather_Avg_HourlyVisibility','origin_weather_Avg_HourlyWin
dGustSpeed','origin_weather_Avg_HourlyWindSpeed','origin_weather_Avg_Precip_Dou
ble','dest_weather_Avg_HourlyVisibility','dest_weather_Avg_HourlyWindGustSpeed'
,'dest_weather_Avg_HourlyWindSpeed','dest_weather_Avg_Precip_Double','pagerank'
,'scheduled_departures_to_destination_prev_4_hours','departures_to_destination_
with_delay_prev_4_hours','frac_departures_to_destination_delayed_prev_4_hours',
'scheduled_origin_departures_prev_4_hours','origin_departures_with_delay_prev_4
_hours','frac_origin_departures_delayed_prev_4_hours', 'min_of_day',
'Airline_Delay_Ranking',
'origin_weather_HourlyPressureTendency_Decreasing','origin_weather_Present_Weat
her_Snow','origin_weather_Present_Weather_SnowGrains','origin_weather_Present_W
eather_IceCrystals','origin_weather_Present_Weather_Hail','origin_weather_Prese
nt_Weather_Mist','origin_weather_Present_Weather_Fog','origin_weather_Present_W
eather_Smoke','origin_weather_Present_Weather_Dust','origin_weather_Present_Wea
ther_Haze','origin_weather_Present_Weather_Storm','dest_weather_HourlyPressureT
endency_Decreasing','dest_weather_Present_Weather_Snow','dest_weather_Present_W
eather_SnowGrains','dest_weather_Present_Weather_IceCrystals','dest_weather_Pre
sent_Weather_Hail','dest_weather_Present_Weather_Mist','dest_weather_Present_We
ather_Fog','dest_weather_Present_Weather_Smoke','dest_weather_Present_Weather_D
ust','dest_weather_Present_Weather_Haze','dest_weather_Present_Weather_Storm']


print(f'Based on EDA, only {len(categoricals)} categorical features should be
kept from the joined dataset.')
print(f'Based on EDA, only {len(numerics)} numeric features should be kept from
the joined dataset.')

Based on EDA, only 9 categorical features should be kept from the joined datase
t.
Based on EDA, only 40 numeric features should be kept from the joined dataset.
```

```
#Create the index, use one hot encoding for the categorical features and impute
the numerics
indexers = map(lambda c: StringIndexer(inputCol=c, outputCol=c+"_idx",
handleInvalid = 'keep'), categoricals)
ohes = map(lambda c: OneHotEncoder(inputCol=c + "_idx",
outputCol=c+"_class"),categoricals)
imputers = Imputer(inputCols = numerics, outputCols = numerics, strategy =
'median')

features = list(map(lambda x: x+'_class', categoricals)) + numerics

#form the stages of the ML pipeline
model_matrix_stages = list(indexers) + list(ohes) + [imputers] + \
                      [VectorAssembler(inputCols=features,
outputCol="ml_features"), StringIndexer(inputCol='DEP_DEL15',
outputCol="label", handleInvalid = 'skip')]

#The kept features are scaled here.  Starting off scaling the features with
unit variance, may switch to z scaler or min/max scaler in the future
scaler = StandardScaler(inputCol="ml_features",
                        outputCol="ml_scaled_features",
                        withStd=True,
                        withMean=True)
```

Due to the unbalanced outcome variable, discovered during EDA, an undersampling technique will be applied to the data to rebalance the outcome variable so that our model does not overfit towards the unbalanced side.

```
def underbalance(df, labelCol):
  """
  function that takes in a data frame and underbalances based on the output
variable
  """
  df_main_reduced = df
  target_year = year
  # Split data by classes
  major_df = df.filter((col(labelCol) == 0))
  minor_df = df.filter((col(labelCol) == 1))

  # Calculate class ratio
  ratio = int(major_df.count()/minor_df.count())
  print("ratio: {}".format(ratio))

  # Sample from major class
  sampled_majority_df = major_df.sample(False, 1/ratio)

  # Combine minor and major classes
  balanced_df = sampled_majority_df.unionAll(minor_df)

  #Sorting data to prevent any data leakage
  balanced_df = balanced_df.withColumn("id",
F.row_number().over(Window.partitionBy().orderBy("FL_DATE")))


  return balanced_df
```

```
print(f'There are {len(features)} features at this stage: \n{features}')
```

```
There are 49 features at this stage:
['QUARTER_class', 'MONTH_class', 'DAY_OF_MONTH_class', 'DAY_OF_WEEK_class', 'OR
IGIN_class', 'DEST_class', 'origin_airport_type_class', 'dest_airport_type_clas
s', 'YEAR_class', 'DISTANCE', 'origin_weather_Avg_HourlyVisibility', 'origin_we
ather_Avg_HourlyWindGustSpeed', 'origin_weather_Avg_HourlyWindSpeed', 'origin_w
eather_Avg_Precip_Double', 'dest_weather_Avg_HourlyVisibility', 'dest_weather_A
vg_HourlyWindGustSpeed', 'dest_weather_Avg_HourlyWindSpeed', 'dest_weather_Avg_
Precip_Double', 'pagerank', 'scheduled_departures_to_destination_prev_4_hours',
'departures_to_destination_with_delay_prev_4_hours', 'frac_departures_to_destin
ation_delayed_prev_4_hours', 'scheduled_origin_departures_prev_4_hours', 'origi
n_departures_with_delay_prev_4_hours', 'frac_origin_departures_delayed_prev_4_h
ours', 'min_of_day', 'Airline_Delay_Ranking', 'origin_weather_HourlyPressureTen
dency_Decreasing', 'origin_weather_Present_Weather_Snow', 'origin_weather_Prese
nt_Weather_SnowGrains', 'origin_weather_Present_Weather_IceCrystals', 'origin_w
```

```
eather_Present_Weather_Hail', 'origin_weather_Present_Weather_Mist', 'origin_we
ather_Present_Weather_Fog', 'origin_weather_Present_Weather_Smoke', 'origin_wea
ther_Present_Weather_Dust', 'origin_weather_Present_Weather_Haze', 'origin_weat
her_Present_Weather_Storm', 'dest_weather_HourlyPressureTendency_Decreasing',
'dest_weather_Present_Weather_Snow', 'dest_weather_Present_Weather_SnowGrains',
'dest_weather_Present_Weather_IceCrystals', 'dest_weather_Present_Weather_Hai
```

Feature Selection with L1 Regularization

Below, we define the linear regression model that we are using for L1 feature selection. The important things to note here is that we set the elasticNetParam to 1 and the regParam to 0.01. The elasticNetParam being equal to 1 specifies that our penalty is purely L1. Specifying an elasticNetParam of 0 means the penalty is purely L2. When we originally set the regParam, we chose a value of 0.1. The reason we changed it to 0.01 was that our first iteration of feature selection eliminated every single feature so we needed to penalize additional features less. This is why we ended up reducing the regularization parameter to 0.01

```
# Defining Regressions With Parameters
lin_reg = LinearRegression(maxIter=10, elasticNetParam=1, regParam = 0.01,
featuresCol = "ml_scaled_features", labelCol = 'label')
```

Next, we define our pipeline for the L1 linear regression and a binary evaluator so we can train a model and evaluate the results.

```
# establish the ML pipeline
pipeline = Pipeline(stages=model_matrix_stages+[scaler])
evaluator = BinaryClassificationEvaluator()
```

Setting up train and test data sets. We will hold out the 2021 data to be used as our final model validation testing. Note that although 2021 data is going to be held out, we will not actualy be testing against this data for this model. This is because the purposes of this model is to simply train it and observe the coefficients so we can select features.

```
# Getting target feature and features list
label = 'DEP_DEL15'
X = categoricals + numerics

#Separating test data set and train dataset with 2021 year being the test data
set
df_lake2 = df_lake.select(X +['FL_DATE'] +[label])
```

We want to make sure that we are training on balanced data. To accomodate this, we will undersample the data set and get a data frame with a ratio of binary outcome variables closer to 1:1. Additionally, since we are only using this model for feature selection, we do not need to use the whole corpus of training data. We sample 0.1% of our training data below. If we did want to use much more this would be infeasible because after our one-hot encoding we have over 1000 features we wish to reduce. Below we will see that only 0.1% of the data still takes 15 minutes for a linear regression model to train.

Finally, we fit our model to the training data and transform our data to include our features column.

```
#Fit and transform
model_pipeline = pipeline.fit(df_lake2)
transformed_df = model_pipeline.transform(df_lake2)

train = df_lake2.filter(transformed_df.YEAR <= 2020).cache()
test = df_lake2.filter(transformed_df.YEAR >2020).cache()
# transformed_df.write.format("delta").mode("overwrite").save(f"
{blob_url}/df_main_transformed_tw1")

#Read From Blob
# transformed_df = spark.read.format( "delta").load(f"
{blob_url}/df_main_transformed001")
```

```
train = transformed_df.filter(transformed_df.YEAR <= 2020).cache()
test = transformed_df.filter(transformed_df.YEAR >2020).cache()
```

```
df_transformed_all_cols = transformed_df.alias('df_transformed_all_cols')
```

For training, we only need our features and target columns so we will drop all others for faster training. We include year and month so as well because this will be used later for cross validation.

```
# Selecting only feature and target columns
transformed_df_for_training = train.select(col('ml_scaled_features'),
col('label'), col('YEAR'), col('MONTH'), col('FL_DATE'))
```

We fit our model below and as we said above, only 1% of the data takes over 15 minutes to train due to the large amount of features we have.

```
# Fitting Linear regression
transformer_lin = lin_reg.fit(transformed_df_for_training)
```

```
transformer_lin.save('/mnt/trainedmodels/lin_reg_model003')

# transformer_lin =
LinearRegressionModel.load('/mnt/trainedmodels/lin_reg_model001')
```

We see the coefficients of our model below. As we hoped, many of these coefficients are equal to zero which means we will be reducing the number of features in our model.

```
# # Showing linear regression coefficients
coefs = transformer_lin.coefficients
coefs
```

```
Out[52]: DenseVector([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
```

```
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
```

Below we get the length of the coefficients and confirm there are 846 features before feature selection.

```
# Number of Coefficients
len(coefs)
```

```
Out[67]: 869
```

Next we check how many features are going to be kept from our feature selection and we can see that there are 13 features that will stay.

```
# Number of kept coefficients
kept = []
for c in coefs:
  if c != 0:
    kept.append(c)

len(kept)
```

```
Out[68]: 3
```

Below we get a list of the feature indices that we would like to keep. This will allow us to slice our original feature vector into a reduced feature vector.

```
# Getting indices of coefficients to keep
indices = []
for i in range(len(coefs)):
  if coefs[i] != 0:
    indices.append(i)

print(indices)

[841, 844, 846]
```

Below we define our vector slicer and specify the input and output columns.

```
# Creating Vector slicer to reduce features
vs = VectorSlicer(outputCol = 'reduced_features', indices = indices)
vs.setInputCol('ml_scaled_features')

Out[56]: VectorSlicer_5d19c73328f9
```

Finally, we use our slicer to transform the dataset and give us a column that contains a reduced feature vector.

```
# Slicing data
transformed_df = vs.transform(transformed_df_for_training)
display(transformed_df)
```

	ml_scaled_features
	▸ {"vectorType": "dense", "length": 869, "values": [-0.5931031586524188, -0.58213156969 1.7540014115599465, -0.5640422663296132, -0.3133175241828996, -0.31310936095303 -0.3068392765687689, -0.30294039856720334, -0.3015946628725108, -0.3008492459 3.343731205046202, -0.2979844721224124, -0.29593983085480713, -0.28975312181205 -0.18578423742712544, -0.18560147451238485, -0.18544202965734394, -0.1853839315 -0.18526817594468464, -0.1851396632353368, -0.18503829071778213, -0.18489357873 -0.18488387191009667, -0.18488038013731062, -0.1847599487829359, -0.18470956256 -0.18454476310926113, -0.18453014383837282, -0.18446955757508282, -0.1843659752 -0.1842608714484223, -0.18423922803371168, -0.18399062564280388, -0.1839800358 -0.18385347860737183, -0.18370226829790479, -0.18329442976678667, -0.1831000740 -0.18308507282505987, -0.18304224656065288, 5.467083744999242, -0.182652535284 -0.17904635390357967, -0.17655813672527082, -0.13685911662075545, -0.4195968736 2.3915938532787506, -0.4177826879622212, -0.4100004679992877, -0.4083867810899 -0.4062678679387189, -0.37685015250640574, -0.25069559255131285, -0.2178243369 -0.20576791839683886, -0.19916525354682604, -0.1842640932565905, -0.1689094598 -0.16442725781578738, -0.15921226698057353, -0.15803235789896977, -0.1564487944 -0.15184443504787326, -0.150402234766657, -0.15020032713795417, -0.145337059689

	-0.14085770300015701, -0.13746744529323696, -0.13668929940470198, -0.1362103954 -0.1278027666287609, -0.12467774398196867, -0.12451497419708907, -0.120374047406 -0.11513551466527615, -0.11403879429315418, -0.11326929749962555, -0.106984211480 -0.10457227147725683, -0.10353335681690469, -0.09698281850985353, -0.096629490

Truncated results, showing first 54 rows.

```
# Writing to blob if not already existed

# # # Save table as Delta Lake
# transformed_df.write.format("delta").mode("overwrite").save(f"
{blob_url}/reduced_featuresDF002")

# # Re-read as Delta Lake in notebook
df_main_reduced = spark.read.format("delta").load(f"
{blob_url}/reduced_featuresDF002")
```

```
df_main_reduced.display()
```

	ml_scaled_features
	▸ {"vectorType": "dense", "length": 846, "values": [-0.5899562704725079, 1.706311369337 -0.5623139225308468, -0.3211961059562208, -0.3173601627195521, -0.31213794400935 3.24290535289039, -0.30604108856257867, -0.2964330430627311, -0.2962693842966 -0.29316908786269924, -0.28754147365235605, -0.28566895327380154, -0.280966564 -0.18903743889640312, -0.18873275470304438, -0.18732016279376867, -0.1873073757 -0.18658153783947726, -0.18620056433822152, -0.18613628884801273, -0.1856729364 -0.18561924302465094, -0.18559131710917384, -0.18513539458138425, -0.18491137807 -0.18478203060772966, -0.18459649539526998, -0.18419250426670278, -0.184093008 -0.18403891553544957, -0.1840280952298417, -0.18396532655161427, -0.18392852246 -0.1830780798454521, -0.18294543140737315, -0.18279963903646107, -0.181791553497 -0.18174342789117168, -0.18112336321028583, 5.557898310955547, -0.179859977101224 -0.175833188725028, -0.13636946228804175, -0.42616739281001054, -0.424279786500 -0.42100503597919237, -0.4073060540070459, 2.4591122240992673, -0.403849044439 -0.3671185554234111, -0.2511004870964505, -0.22639788143186113, -0.2081010286749 -0.2000761727850778, -0.19076864551977868, -0.1695520656825809, -0.16464913603C -0.16335062712225848, -0.16193379337669767, -0.15704275787155755, -0.14754223093 -0.14654254074565365, -0.14649212693753222, -0.1456966686259268, -0.1455579213E -0.14518109110603641, -0.14481677147712835, -0.1321822580598325, -0.128845452518 -0.12867200448812188, -0.1254626150661146, -0.12354924183297798, -0.12310084950 -0.12074651864399494, -0.1147850933228211, -0.11291126157658408, -0.111580446033E -0.10628167680725216, -0.1027957845517992, -0.09962088267329373, -0.09886279527

Truncated results, showing first 55 rows.

Now that we have all the features selected from our L1 regression, we are interested in determining which features have made it throught the feature selection process. To do this by first compiling a list of feature names that correspond to the feature indices of our dense vector. We do this below by looping through every feature name in order. If the feature is categorical, we use our helper function to append an ordered list of the one hot encodings. If the feature is numeric, we append the numeric feature name and this will give us a list of feature names that correspond to the feature indices we calculated above. The output of this cell observes the order of

```
def get_sparse_feature_names(feature, idx, df):
  """Helper function that returns a list of ordered feature names for
categorical variables
  Inputs: feature - feature name like ORIGIN
  idx - feature index name created by pipeline like ORIGIN_idx
  df - main dataframe"""

  # gets unique feature index pairs, sorts by index, maps categorical value to
name, returns list
  selected_df = df.select([feature, idx]).distinct()
  selected_df = selected_df.orderBy(idx)
  selected_df = selected_df.toPandas()
  selected_df[feature] = selected_df[feature].map(lambda x: str(feature) + '_'
+ str(x))
  return list(selected_df[feature])

# getting all columns added by pipeline and original columns before pipeline
added columns
all_columns = df_transformed_all_cols.columns
original_columns = df_transformed_all_cols.columns[:49]

# initializing list of feature names that correspond to feature indices
final_features = []

# looping thru each feature
for col in original_columns:
  # If categorical
  if col + '_idx' in all_columns:
    # use helper function to get list of one hot encoded variable names, add to
final feature list
    feature = col
    idx = feature + '_idx'
    features = get_sparse_feature_names(feature, idx, df_transformed_all_cols)
    final_features = final_features + features
  else:
    # if numeric, add feature name to list
    final_features.append(col)

final_features
```

```
Out[72]: ['QUARTER_3',
 'QUARTER_4',
 'QUARTER_1',
 'QUARTER_2',
 'MONTH_7',
 'MONTH_8',
```

```
 'MONTH_3',
 'MONTH_10',
 'MONTH_12',
 'MONTH_6',
 'MONTH_11',
 'MONTH_1',
 'MONTH_9',
 'MONTH_5',
 'MONTH_4',
 'MONTH_2',
 'DAY_OF_MONTH_19',
 'DAY_OF_MONTH_12',
 'DAY_OF_MONTH_22',
 'DAY_OF_MONTH_20',
```

Finally, to view the selected feature names, we select from our ordered feature list, the indices of the features we are selecting. The output is shown below.

```
# Initializing list of selected features
selected_features = []

# looping thru features and pulling out selected features by index
for i in range(len(final_features)):
  if i in indices:
    selected_features.append(final_features[i])

# viewing selected features
selected_features

Out[73]: ['frac_departures_to_destination_delayed_prev_4_hours',
 'frac_origin_departures_delayed_prev_4_hours',
 'Airline_Delay_Ranking']
```

Splitting Data into Training and Test Sets and Cross Validation

We want to use blocked time series cross validation to further enhance our model. Since our input is time-series data, we must be careful to prevent any data leakage which could allow the model to train on "future" data to predict the present. To do so, we will use 6 folds, one for each year 2015 through 2020, using December as our validation chunk for each fold.

Step 6: Build Logistic Regression Model

The team will run a logistic regression model to predict whether or not the flight will be delayed.

For logistic regression, the team plans to use Log Loss as this is the standard practice when running logistic classification problems and will help visualize the results later on. The team also intends to use a hybrid approach to regularization and will use the elastic net regularization method , which aims to minimize the amount of strongly correlated variables as well as making the loss function strongly convex.

Helper functions for training models and running cross validation on time-series

```
def build_ml_pipeline(params):
  """
  creates a pipeline to transform dataset into a machine learning ready
dataframe
  """

  categoricals = ['origin_airport_type','dest_airport_type']

  numerics = ['DISTANCE', 'origin_weather_Avg_HourlyWindSpeed',
'dest_weather_Avg_HourlyWindSpeed','scheduled_departures_to_destination_prev_4_
hours',

'frac_departures_to_destination_delayed_prev_4_hours','scheduled_origin_departu
res_prev_4_hours',
              'frac_origin_departures_delayed_prev_4_hours', 'min_of_day',
'Airline_Delay_Ranking']

  #Create the index, use one hot encoding for the categorical features and
impute the numerics
  indexers = map(lambda c: StringIndexer(inputCol=c, outputCol=c+"_idx",
handleInvalid = 'keep'), categoricals)
  ohes = map(lambda c: OneHotEncoder(inputCol=c + "_idx",
outputCol=c+"_class"),categoricals)
  imputers = Imputer(inputCols = numerics, outputCols = numerics, strategy =
'median')

  features = list(map(lambda x: x+'_class', categoricals)) + numerics

  #form the stages of the ML pipeline
  model_matrix_stages = list(indexers) + list(ohes) + [imputers] + \
                        [VectorAssembler(inputCols=features,
outputCol="ml_features"), StringIndexer(inputCol='DEP_DEL15',
outputCol="label", handleInvalid = 'keep')]


#   #Starting off scaling the features with unit variance ----- may switch to z
scaler or min/max scaler in the future after more experimentation
  scaler = StandardScaler(inputCol="ml_features",
                          outputCol="ml_scaled_features",
                          withStd=True, withMean=True)


    # establish the ML pipeline
  pipeline = Pipeline(stages=model_matrix_stages +  [scaler])
```

```
    return pipeline
```

```
def cross_val(df, grid, model_type, k=6):
  """
  custom cross validation function that takes in training data and runs a k
fold cross validation using the model type provided
  """
  #Helper functions
  def set_model_params(grid):
    params = grid.values()
    param_names = list(grid.keys())
    param_values = list(itertools.product(*params))
    return(param_names, param_values)

  # Initiate trackers
  top_score = 0
  top_params = None

  scores = {}
  n=df.count()
  kfold_size = int(n/(k+1))

  p_names, params = set_model_params(grid)
  df = df.withColumn("id",
F.row_number().over(Window.partitionBy().orderBy("FL_DATE")))

  #Walk through each parameter combination
  for param in params:

    pipe = build_ml_pipeline(param)

    #select which model to run
    if model_type == 'xgb':
      built_model = run_xgb(param)

    elif model_type =='log_reg':
      built_model = run_lr(param)

  #walk though each fold and run xgBoost grid search on each
    fold_score = []
    for num in range(k):
      print(f'Starting fold {num} with parameters {list(zip(p_names, param))}')
      train = df.filter((F.col('id') <= kfold_size * (num+1))&(F.col('id') >
kfold_size*num))
      dev = df.filter((F.col('id') > kfold_size * (num+1))&(F.col('id')<
kfold_size * (num +2)))

      print('training started')
```

```
      model = built_model.fit(train)
      print('training ended')
      preds = model.transform(dev)
      print('predicting done')

      pred_results = preds.select('label', 'pred').rdd

      if model_type == 'xgb':
        auc = BinaryClassificationMetrics(pred_results.map(lambda x:
(float(x.label), float(x.pred[1])))).areaUnderROC

      elif model_type =='log_reg':
        auc = BinaryClassificationMetrics(pred_results.map(lambda x:
(float(x.label), float(x.pred)))).areaUnderROC

      print(f'The AUC score for fold {num} is: {auc} using the parameters:
{list(zip(p_names, param))}')

      #add this score to the dictionary
      scores[param] = auc
      #add this score to fold score for averaging later
      fold_score.append(auc)

    #calc_average
    average_fold_score = np.average(fold_score)
    print(f'The average score for the parameters: {list(zip(p_names, param))}
is {average_fold_score}')

    if average_fold_score > top_score:
      top_score = average_fold_score
      top_params = list(zip(p_names, param))

  print(f"The top AUC score for this CV run was {top_score}")
  return top_score, top_params
```

```
def run_lr(params):
  """
  Runs LR model given the provided parameters
  """
  log_reg = LogisticRegression(maxIter=10, elasticNetParam=params[1],
featuresCol = "reduced_features", labelCol = 'label', predictionCol = 'pred',
                               regParam = params[0])

  return log_reg



# Logistic regression model
log_grid = {'regParam': [0.01, 0.05, 0.1],
           'elasticNetParam': [0.5, 1]}



df_train = df_main_reduced.select('reduced_features', 'label', 'FL_DATE')

best_lr_score, best_lr_params = cross_val(df_train, log_grid, model_type =
'log_reg', k=6)
```

```
ram', 0.05), ('elasticNetParam', 1)]
Starting fold 1 with parameters [('regParam', 0.05), ('elasticNetParam', 1)]
training started
training ended
predicting done
The AUC score for fold 1 is: 0.8050359710489302 using the parameters: [('regPa
ram', 0.05), ('elasticNetParam', 1)]
Starting fold 2 with parameters [('regParam', 0.05), ('elasticNetParam', 1)]
training started
training ended
predicting done
The AUC score for fold 2 is: 0.8221963926620061 using the parameters: [('regPa
ram', 0.05), ('elasticNetParam', 1)]
Starting fold 3 with parameters [('regParam', 0.05), ('elasticNetParam', 1)]
training started
training ended
predicting done
The AUC score for fold 3 is: 0.814235256753796 using the parameters: [('regPar
am', 0.05), ('elasticNetParam', 1)]
Starting fold 4 with parameters [('regParam', 0.05), ('elasticNetParam', 1)]
training started
training ended
```

Time and compute limitations resulted in an early stoping of the above grid search with one parameter combination still to run. The best average AUC score up until this point was: 0.825

Step 7: Build XGBoost Model

Running an XGBoost model on the reduced features. We wanted to tune some of the many hyperparameters of the XGBoost algorithm.

```
def run_xgb(params):
  """
  Runs xgb model given the provided parameters
  """
  xgboost = XgboostClassifier(featuresCol = 'reduced_features', labelCol =
'label', rawPredictionCol = 'pred',
                              missing = 0.0, eval_metric = 'auc',
                              max_depth=params[0],
                              n_estimators=params[1],
                              reg_lambda=params[2],
                              reg_alpha=params[3],
                              objective=params[4],
                              base_score = params[5],
                              learning_rate = params[6],
                              gamma = params[7],
                              scale_pos_weight = params[8],
                              min_child_weight = params[9])

  return xgboost


xgb_grid = {'max_depth':[2,6],
                      'n_estimators':[20, 100],
                      'reg_lambda':[1],
                      'reg_alpha': [0],
                      'objective':['binary:logistic'],
                      'base_score':[0.5],
                      'learning_rate':[0.2, 0.3],
                      'gamma':[0.05, 0.1],
                      'scale_pos_weight':[1],             #we want to use a
ratio of outcome variables, since we already balanced, this should be set to 1
                      'min_child_weight':[1.5]}
```

```
df_train = df_main_reduced.select('reduced_features', 'label', 'FL_DATE')
best_xgb_score, best_xgb_params = cross_val(df_train, xgb_grid, model_type =
'xgb',k=6)
```

```
Starting fold 0 with parameters [('max_depth', 2), ('n_estimators', 20), ('reg
_lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_scor
e', 0.5), ('learning_rate', 0.2), ('gamma', 0.05), ('scale_pos_weight', 2),
('min_child_weight', 1.5)]
training started
training ended
predicting done
The AUC score for fold 0 is: 0.746920834145141 using the parameters: [('max_de
pth', 2), ('n_estimators', 20), ('reg_lambda', 1), ('reg_alpha', 0), ('objecti
ve', 'binary:logistic'), ('base_score', 0.5), ('learning_rate', 0.2), ('gamm
a', 0.05), ('scale_pos_weight', 2), ('min_child_weight', 1.5)]
Starting fold 1 with parameters [('max_depth', 2), ('n_estimators', 20), ('reg
_lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_scor
e', 0.5), ('learning_rate', 0.2), ('gamma', 0.05), ('scale_pos_weight', 2),
('min_child_weight', 1.5)]
training started
training ended
predicting done
The AUC score for fold 1 is: 0.7522388211687623 using the parameters: [('max_d
epth', 2), ('n_estimators', 20), ('reg_lambda', 1), ('reg_alpha', 0), ('object
ive', 'binary:logistic'), ('base_score', 0.5), ('learning_rate', 0.2), ('gamm
```

Running an experiment to see if our pagerank feature, which dropped out during L1 regression, will add any improvement to our model.

```
indices = [841, 844, 846]

page_rank_index = [845]

# Creating Vector slicer to reduce features
vs = VectorSlicer(outputCol = 'reduced_features', indices = indices +
page_rank_index)
vs.setInputCol('ml_scaled_features')

# Slicing data
df_main_reduced_pr = df_main_reduced.select('ml_scaled_features', 'label',
'FL_DATE')
transformed_df_pr = vs.transform(df_main_reduced)


def run_xgb(params):
  """
  Runs xgb model given the provided parameters
  """
  xgboost = XgboostClassifier(featuresCol = 'reduced_features', labelCol =
'label', rawPredictionCol = 'pred',
                              missing = 0.0,
                              max_depth=params[0],
                              n_estimators=params[1],
                              reg_lambda=params[2],
                              reg_alpha=params[3],
                              objective=params[4],
                              base_score = params[5],
                              learning_rate = params[6],
                              gamma = params[7],
                              scale_pos_weight = params[8],
                              min_child_weight = params[9])

  return xgboost

xgb_grid = {'max_depth':[6],
                      'n_estimators':[100],
                      'reg_lambda':[1],
                      'reg_alpha':[0],
                      'objective':['binary:logistic'],
                      'base_score':[0.5],
                      'learning_rate':[0.3],
                      'gamma':[0.05],
                      'scale_pos_weight':[2],
                      'min_child_weight':[1.5]}
```

```
df_train = transformed_df.select('reduced_features', 'label', 'FL_DATE')
best_xgb_PR_score, best_xgb_PR_params = cross_val(df_train, xgb_grid,
model_type = 'xgb',k=6)
```

```
Starting fold 0 with parameters [('max_depth', 6), ('n_estimators', 100), ('re
g_lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_scor
e', 0.5), ('learning_rate', 0.3), ('gamma', 0.05), ('scale_pos_weight', 2),
('min_child_weight', 1.5)]
training started
training ended
predicting done
/databricks/spark/python/pyspark/sql/context.py:134: FutureWarning: Deprecated
in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
The AUC score for fold 0 is: 0.7563745977858934 using the parameters: [('max_d
epth', 6), ('n_estimators', 100), ('reg_lambda', 1), ('reg_alpha', 0), ('objec
tive', 'binary:logistic'), ('base_score', 0.5), ('learning_rate', 0.3), ('gamm
a', 0.05), ('scale_pos_weight', 2), ('min_child_weight', 1.5)]
Starting fold 1 with parameters [('max_depth', 6), ('n_estimators', 100), ('re
g_lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_scor
e', 0.5), ('learning_rate', 0.3), ('gamma', 0.05), ('scale_pos_weight', 2),
('min_child_weight', 1.5)]
training started
training ended
predicting done
```

adding in the page rank algorithm did seem to increase our best average AUC, however, the increase was minimal

Running the xgb algorithm with double the folds to experiement with results

```
xgb_grid = {'max_depth':[6],
                      'n_estimators':[20],
                      'reg_lambda':[1],
                      'reg_alpha':[0],
                      'objective':['binary:logistic'],
                      'base_score':[0.5],
                      'learning_rate':[0.3],
                      'gamma':[0.05],
                      'scale_pos_weight':[2],
                      'min_child_weight':[1.5]}

df_train = transformed_df.select('reduced_features', 'label', 'FL_DATE')
best_xgb_12k_score, best_xgb_12k_params = cross_val(df_train, xgb_grid,
model_type = 'xgb',k=12)
```

```
Starting fold 0 with parameters [('max_depth', 6), ('n_estimators', 20), ('reg_
lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_score',
0.5), ('learning_rate', 0.3), ('gamma', 0.05), ('scale_pos_weight', 2), ('min_c
hild_weight', 1.5)]
training started
  Cancelled
```

```
#result above scored an AUC of .75, not significantly different than the other
xgb validation runs
```

Step 8: Model Evaluation

Our team is focusing on optimizing two success metrics. First, the team believes that Area Under the Curve - Receiver Operating Characteristics (AUC-ROC) is a good metric to use as it works well with classification algorithms, especially with unbalanced outcome variables such as ours. AUC-ROC curve can be calculated using the following two equations to determine the axis and plotting the ROC curve. True Positive Rate = TP/(TP+FN) (Eq. 4)

```
                                        False Positive Rate = FP/(TN+FP)
```

The secondondary metric the team has selected is the false negative rate (equation 6). When helping our aviation logistics customers predict flight delays, it is much more important to reduce the amount of false negatives the model predicts so that

oue logistic consumers are able to optimize their ability to shift their operation and meet customer needs.

```
                                        False Negative Rate = FN/(FN+TP)
```

The results from the 6-fold cross validation were encouraging, but also presented us with additional puzzles for further exploration. Across Folds 1-5 (years 2015-2019) the results were fairly consistent–even if flights are balanced 50%/50% between delayed flights and non-delayed flights, we are able to predict whether or not they will be delayed with 69% AUC. However, we AUC actually increased to 75% in 2021, which was a peak covid-impact year. We suspect that the decline in airline travel may have left airlines with excess capacity, which meant that instead of congestion-driven delays or other indirect variables which are more difficult to measure, delays were more directly driven by other variables such as weather, which we were able to predict more accurately. We intend to research these further in future years.

Transforming our complete datasets for final eval

```
vs = VectorSlicer(outputCol = 'reduced_features', indices = indices)
vs.setInputCol('ml_scaled_features')

test_df = vs.transform(test)
test_df = test_df.select('reduced_features', 'label', 'FL_DATE')
```

```
def run_lr(params):
  """
  Runs logistic regression model given the provided parameters
  """
  log_reg = LogisticRegression(maxIter=10, elasticNetParam=0.5, featuresCol =
"reduced_features", labelCol = 'label', predictionCol = 'pred',
                               regParam = params[0])

  return log_reg
```

```
def run_xgb(params):
  """
  Runs xgb model given the provided parameters
  """
  xgboost = XgboostClassifier(featuresCol = 'reduced_features', labelCol =
'label', rawPredictionCol = 'pred',
                              missing = 0.0, eval_metric = 'auc',
                              max_depth=params[0],
                              n_estimators=params[1],
                              reg_lambda=params[2],
                              reg_alpha=params[3],
                              objective=params[4],
                              base_score = params[5],
                              learning_rate = params[6],
                              gamma = params[7],
                              scale_pos_weight = params[8],
                              min_child_weight = params[9])

  return xgboost
```

```
def test_eval(train_df, test_df, grid, model_type, k=1):
  """
  function that takes in full train and test set (2021) and outputs the final
evaluation score
  """
  #Helper functions
  def set_model_params(grid):
    params = grid.values()
    param_names = list(grid.keys())
    param_values = list(itertools.product(*params))
    return(param_names, param_values)

  # Initiate trackers
  top_score = 0
  top_params = None

  scores = {}

  p_names, params = set_model_params(grid)

  #Walk through each parameter combination
  for param in params:

    pipe = build_ml_pipeline(param)

    #select which model to run
    if model_type == 'xgb':
      built_model = run_xgb(param)

    elif model_type =='log_reg':
      built_model = run_lr(param)

  #walk though each fold and run xgBoost grid search on each
    fold_score = []
    for num in range(k):
      print(f'Starting fold {num} with parameters {list(zip(p_names, param))}')
      train = train_df
      dev = test_df

      print('training started')
      model = built_model.fit(train)
      print('training ended')
      preds = model.transform(dev)
      print('predicting done')

      pred_results = preds.select('label', 'pred').rdd
```

```
      if model_type == 'xgb':
        auc = BinaryClassificationMetrics(pred_results.map(lambda x:
(float(x.label), float(x.pred[1])))).areaUnderROC

      elif model_type =='log_reg':
        auc = BinaryClassificationMetrics(pred_results.map(lambda x:
(float(x.label), float(x.pred)))).areaUnderROC

      print(f'The AUC score for fold {num} is: {auc} using the parameters:
{list(zip(p_names, param))}')

      #add this score to the dictionary
      scores[param] = auc
      #add this score to fold score for averaging later
      fold_score.append(auc)

    #calc_average
    average_fold_score = np.average(fold_score)
    print(f'The average score for the parameters: {list(zip(p_names, param))}
is {average_fold_score}')

    if average_fold_score > top_score:
      top_score = average_fold_score
      top_params = list(zip(p_names, param))

  print(f"The top AUC score for this CV run was {top_score}")
  return top_score, model
```

```
final_xgb_params = {'max_depth':[6],
                    'n_estimators':[100],
                    'reg_lambda':[1],
                    'reg_alpha':[0],
                    'objective':['binary:logistic'],
                    'base_score':[0.5],
                    'learning_rate':[0.3],
                    'gamma':[0.05],
                    'scale_pos_weight':[2],
                    'min_child_weight':[1.5]}

# Logistic regression model
final_lr_params = {'regParam': [0.05]}
```

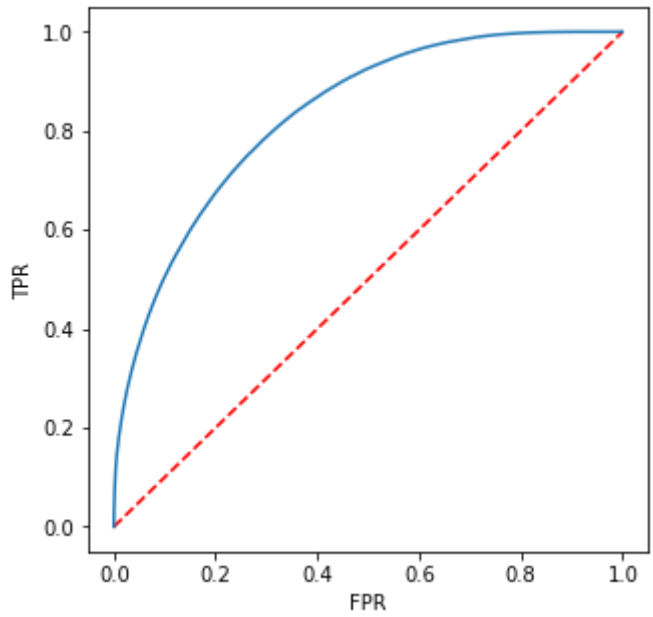
```
train_df = transformed_df.sample(.3, 2022) #sampling to reduce vary large
compute time
train_df = train_df.select('reduced_features', 'label', 'FL_DATE')

lr_final_score, lr_final_model = test_eval(train_df, test_df, final_lr_params,
'log_reg')
```

```
Starting fold 0 with parameters [('regParam', 0.05)]
training started
training ended
predicting done
/databricks/spark/python/pyspark/sql/context.py:134: FutureWarning: Deprecated
in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
The AUC score for fold 0 is: 0.8065063097532388 using the parameters: [('regPar
am', 0.05)]
The average score for the parameters: [('regParam', 0.05)] is 0.806506309753238
8
The top AUC score for this CV run was 0.8065063097532388
```

```
lr_final_model.save('/mnt/trainedmodels/lr_final_model')
```

```
plt.figure(figsize=(5,5))
plt.plot([0, 1], [0, 1], 'r--')
plt.plot(lr_final_model.summary.roc.select('FPR').collect(),
         lr_final_model.summary.roc.select('TPR').collect())
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```




```
preds = lr_final_model.transform(test_df)
pred_results = preds.select('label', 'pred').rdd
```

```
train_df = transformed_df.sample(.3, 2022) #sampling to reduce vary large
compute time
train_df = train_df.select('reduced_features', 'label', 'FL_DATE')

xgb_final_score, xgb_final_model = test_eval(train_df, test_df,
final_xgb_params, 'xgb')
```

```
Starting fold 0 with parameters [('max_depth', 6), ('n_estimators', 20), ('reg_
lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_score',
0.5), ('learning_rate', 0.3), ('gamma', 0.05), ('scale_pos_weight', 2), ('min_c
hild_weight', 1.5)]
training started
training ended
predicting done
The AUC score for fold 0 is: 0.7597734875147854 using the parameters: [('max_de
pth', 6), ('n_estimators', 20), ('reg_lambda', 1), ('reg_alpha', 0), ('objectiv
e', 'binary:logistic'), ('base_score', 0.5), ('learning_rate', 0.3), ('gamma',
0.05), ('scale_pos_weight', 2), ('min_child_weight', 1.5)]
The average score for the parameters: [('max_depth', 6), ('n_estimators', 20),
('reg_lambda', 1), ('reg_alpha', 0), ('objective', 'binary:logistic'), ('base_s
core', 0.5), ('learning_rate', 0.3), ('gamma', 0.05), ('scale_pos_weight', 2),
('min_child_weight', 1.5)] is 0.7597734875147854
The top AUC score for this CV run was 0.7597734875147854
```

```
xgb_final_model.save('/mnt/trainedmodels/xgb_final_model')
```

Out[111]:

	Model	AUC
0	Baseline	0.500000
1	Logistic Regression	0.806506
2	XGBoost	0.759773