databricksFP_Section2_Group2_Phase4_MASTER

(https://databricks.com)

FP_Section2_Group2_Phase4_MASTER

DATABRICKS URL: https://adb-731998097721284.4.azuredatabricks.net/?o=731998097721284#notebook/942460095360751/command/942460095360752 (https://adb-731998097721284.4.azuredatabricks.net/?o=731998097721284#notebook/942460095360751/command/942460095360752)

Team and project meta information

Team Information

Project Title: Predicting Flight Delays to Reduce Airline Economic Loss
Group Number: Section 2 Group 2
Team Name: Mars
Team Members Morgan Raymond Fang, Audrey Phone, Sophey Yeh, Morgan Yung

Contact Information

	Name	Email
1	Audrey Phone	audreyphone@berkeley.edu (mailto:audreyphone@berkeley.edu)
2	Raymond Fang	rafang25@berkeley.edu (mailto:rafang25@berkeley.edu)
3	Morgan Yung	my4223@berkeley.edu (mailto:my4223@berkeley.edu)
4	Sophie Yeh	syeh1@berkeley.edu (mailto:syeh1@berkeley.edu)

Credit Assignments

Task	Assignment	Assignee	Start Date	Due Date	Time Allotment	Priority
2	EDA	Group	11-28	11-29	240 Min	H
3	Feature Engineering	Group	11-29	11-30	420 Min	H
4	Data Lineage	Raymond F.	11-29	11-30	120 Min	M

Task	Assignment	Assignee	Start Date	Due Date	Time Allotment	Priority
5	Neural Network P1	Sophie Y.	11-30	12-1	120 Min	M
6	Grid Search Implementation	Group	11-30	12-1	60 Min	H
7	Build Evaluate Functions	Sophie Y.	11-30	12-1	180 Min	H
8	Implement Early Stop	Morgan Y.	12-2	12-3	300 Min	M
9	Decide Novel direction	Ray. F & Audrey P.	12-2	12-3	30 Min	M
10	SMOTE	Ray. F & Audrey P.	12-2	12-3	720 Min	H
11	Experiments	Group	12-2	12-3	180 Min	M
12	Develop Neural Network Architecture	Sophie Y.	12-2	12-3	120 Min	M
13	Update Project Leaderboard w/ results	Sophie Y.	12-3	12-4	15 Min	L
14	Clean up Code	Group	12-3	12-4	240 Min	H

Thesis

Many factors impact the predition of flight delays. Features ranging from weather, time, and flight related features all play a role in determining the outcome of a flight delay.

Abstract

Flight delays have been an increasingly common problem for airlines over the last decade. These delays can lead to cascading effects of additional delays, and even worse flight cancellations. Being able to better predict flight delays will allow operational optimizations and a reduction in economic losses. The goal of our project is to create a machine learning model that predicts flight departure delays 2 hours ahead of time as a tool for airlines to improve flight scheduling and better respond to such delays. The focus of Phase 4 is to introduced more complex models as well as additional features that would help improve the predictive power of our model. In this phase, we introduced four new features, developed a multilayered neural network, and applied SMOTE to handle our feature imbalance concerns. Our results indicate that MNN performed the best with a test weighted F1 score of 0.7651.

Introduction

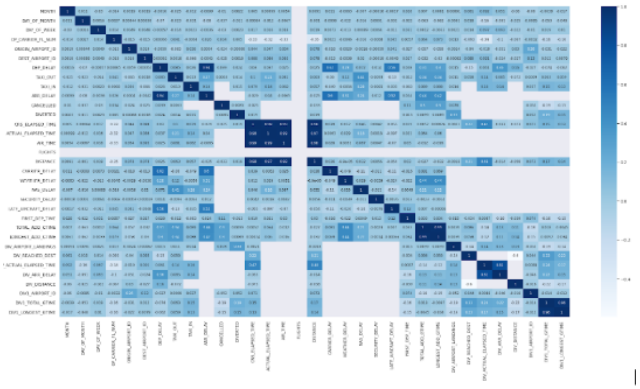
Business Case

Flight delays have been an increasingly common problem for airlines over the last decade. These delays can lead to cascading effects of additional delays, and even worse flight cancellations. Being able to better predict flight delays will allow operational optimizations and a reduction in economic losses. The goal of our project is to create a machine learning model that will aid in predicting flight departure delays 2 hours ahead of time as a tool for airlines to improve flight scheduling and better respond to such delays. We define delays as flights departing more than 15 minutes later than the scheduled time. By providing insights on flight delays, we hope to improve overall customer satisfaction, leading to a higher rate of returning customers and revenue for airlines. The target focus of this study are the customers purchasing flights from Airlines. We will measure the success of our Machine Learning algorithm by measuring the F1 score.

Datasets

Flights Dataset

The flight dataset was downloaded from the US Department of Transportation and contains flight information from 2015 to 2021. The full airlines data has ~74M rows and 111 columns. The dataset consists of flights that were flown across the United States from 2014 - 2021. The flights are evenly distributed across time, but there are major class imbalances when it comes to delayed versus non-delayed flights. There are about 4.8x more non-delayed flights compared to delayed flights. The label in question, DEP_DEL15 also does not seem to have much correlation with most other features in this dataset. We will be relying on newly introduced features, feature engineered or otherwise.



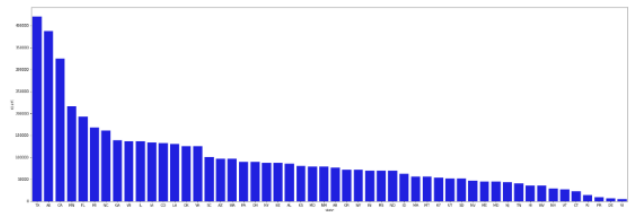
Stations

The dataset displays a list of all stations with each record identify the station and it's distance relative to its neighbor. Overall, there are 2,202 unique station names spread across North America, but based on their latitude and longitude, there are 2,225 unique stations.

While the dataset did provide us with a handful of useful information on the stations, some information was only provided to the neighboring stations such as the state and call IDs. To populate this information, a filter was placed on the dataset where the distance_to_neighbor was equal to zero, implying the neighbor is itself. The data is then joined back on the stations dataset to provide a more wholistic picture of where stations resided.

Based on the map of North America below, the states on the central - eastern regions of the United States have a much denser population of stations compared to that of the western side of the country.

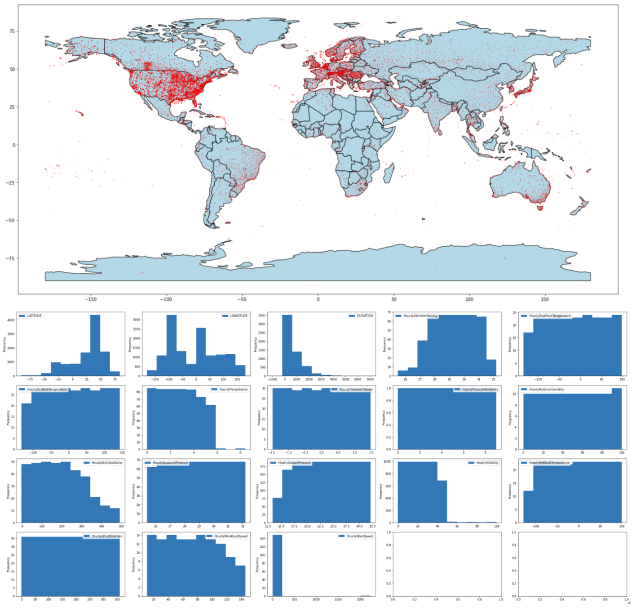
However, that is not to say there is a weak presence on the western side of the country. On the contrary, of the top five states with the most stations, two of them are on the western side, Alaska and California. They simply are not as densely populated with stations due to the sheer size of the territories.



Weather

Geoplot shows that weather data set includes data from all around the world. Our primary focus though will only be on weather data in the US. The weather dataset has many null values for various different columns. The null columns varied and some observations had varying different columns that were null and were inconsistent. Not all data followed a normal distribution. Analysis was primarily done on a subset based on prior knowledge. Hourly measurement data was the primary variables looked at for the exploratory data analysis since flights happen on an hourly basis. Histograms were used to identify inaccurate values as well as analyze the distribution of variables as shown below.

Relative Humidity in particular did not follow a normal distribution. Wind direction was disregarded due to being a more ordinal variable. Of these variables, elevation and hourly wind speed had inaccurate values that will be filtered out in the final data set. The distribution of wind speed followed a normal distribution after removing aforementioned values as shown in the image below.



Final Dataset

We took the three datasets above and joined them together, using the Flights dataset as our base. We proceeded to introduce additional features ranging from Time Series features such as time of the Flights as well as Graph Based features like Page Rank. We introduced features such as event based (holidays), nature based (natural disasters) and airport related (reputation & maintenance) features. We also reduced the number of features by means of a PCA and reduced the dataset by removing any cancelled flights and some null records.

The final dataset came out to a dataset with ~41M records and 172 features. The final dataset will eventually be reduced further to the final features that provided the most predicting power.

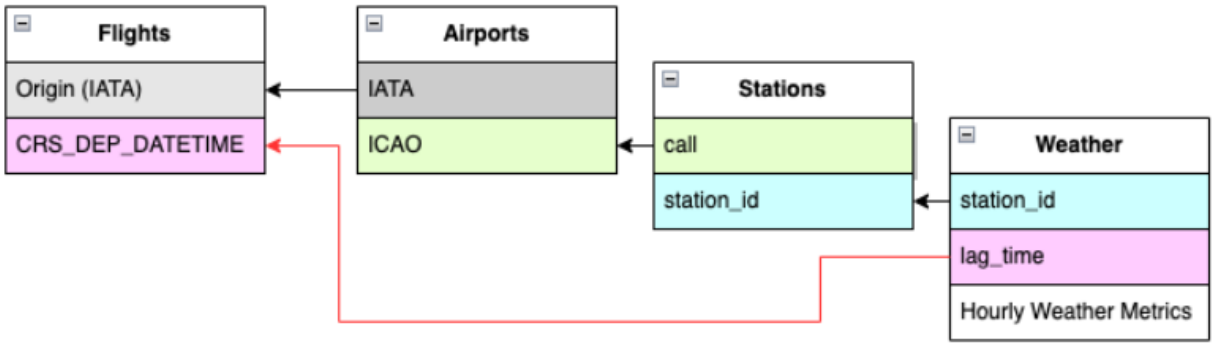
Imports & Setup

```
# Python packages

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
from datetime import timedelta
pd.set_option('max_columns', None)
import time
# Pyspark
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.window import Window
from pyspark.sql.functions import col,isnan,when,count,array
from pyspark.sql.types import IntegerType
from pyspark.sql.types import TimestampType, DateType, StringType, NumericType
from pyspark.ml.stat import Correlation
from pyspark.sql import Window as W
from pyspark import SparkConf, SparkContext
import random
import numpy as np
from functools import reduce
from pyspark.sql import Row
from pyspark.sql.functions import rand,col,when,concat,substring,lit,udf,lower,sum as ps_sum,count as ps_count,row_number
from pyspark.sql.window import *
from pyspark.sql import DataFrame
from pyspark.ml.feature import
VectorAssembler,BucketedRandomProjectionLSH,VectorSlicer,StringIndexer,OneHotEncoder,StandardScaler
from pyspark.ml.linalg import Vectors,VectorUDT, SparseVector
from pyspark.sql.functions import array, create_map, struct
from pyspark.ml import Pipeline
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator
from imblearn.over_sampling import SMOTE
from imblearn.combine import SMOTEENN

blob_container = "w261" # The name of your container created in https://portal.azure.com
storage_account = "syeh" # The name of your Storage account created in https://portal.azure.com
secret_scope = "w261" # The name of the scope created in your local computer using the Databricks CLI
secret_key = "w261" # saskey 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"

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


file = f"joined_df_time/"
df_past = spark.read.parquet(f"{blob_url}/{file}")
df_2022 = spark.read.parquet(f"{blob_url}/flights_2022/")
df_2014 = spark.read.parquet(f"{blob_url}/flights_2014/")
```

Data Preparation

Data Lineage

We start our process by intaking the datasets mentioned above and create a joined dataset. As mentioned before, the flights dataset is the core for our final dataset. We joined Flights to Airport (Oirgin <> IATA), Airports to Stations (ICAO <> call), Weather to Stations (station_id <> station_id), and finally Weather back to Flights (CRS_DEP_DATETIME <> lag_time)



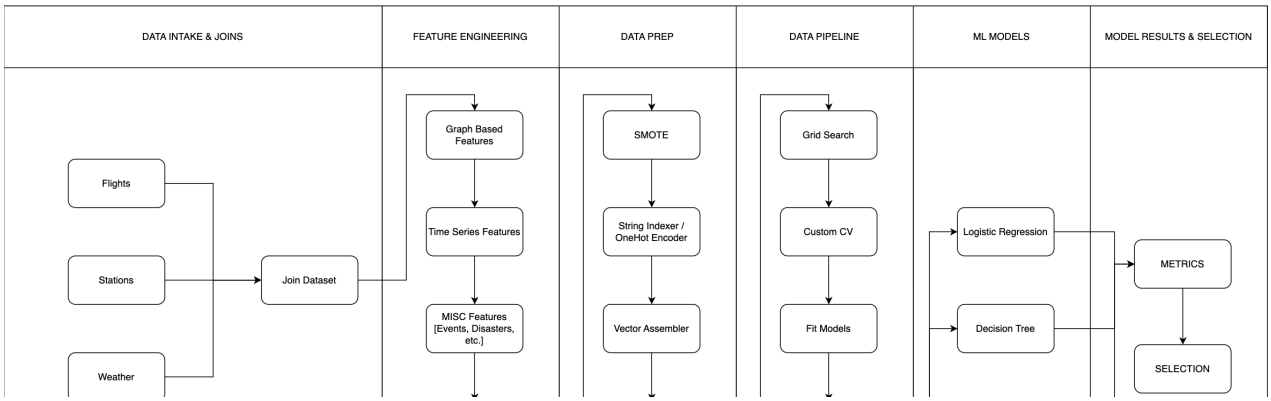
We took this dataset and removed any flights that would not be valid in our analysis, primarily cancelled flights.

We then perform EDA on the joined dataset. While we went over this in detail in Phases 1 - 3, we opted to perform additional EDA on flights that took place in 2022. We were curious if there were any trends from the start of the COVID-19 pandemic and if there were any trends that persisted through 2022.

Next, we took our findings from the EDA and perform our feature engineering step(s). This step includes adding in features including: graph based, time-series based, and miscellaneous features such as events taking place, natural disasters, and historical features of the airport (reputation, history of maintenance delays, etc). The dataset is then processed further by applying dimension reduction techniques like a PCA. An additional year of data (2014) was also added in to supplement some of the feature engineer process.

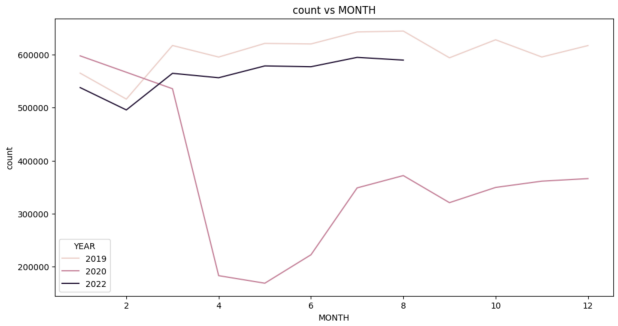
The next step is to take this curated dataset and enter our Data Preperation phase. A SMOTE technique is applied to handle class imbalance issues. To handle categorical features, they will be passed through a string indexer and then one-hot encoded. After this is complete, the features will be passed through a vector assembler and scaled accordingly.

Finally, we will pass the curated dataset through ML three models: Logistic Regression, Decision Tree, and a Multi-Layer Neural Network (MNN). The metric we will be scoring these models on is the F1 Score.

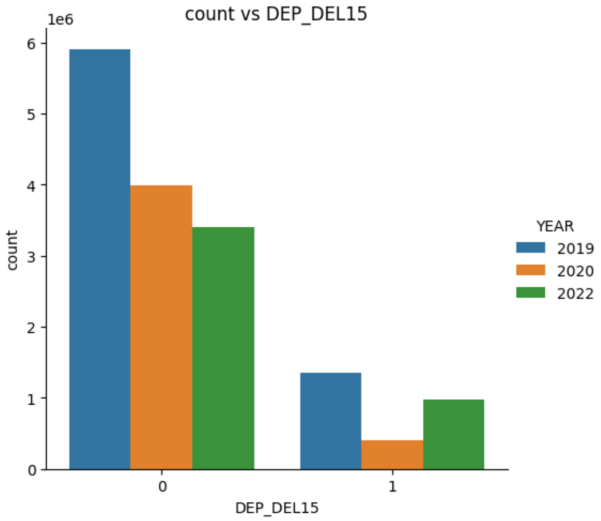


2019 - 2022 EDA

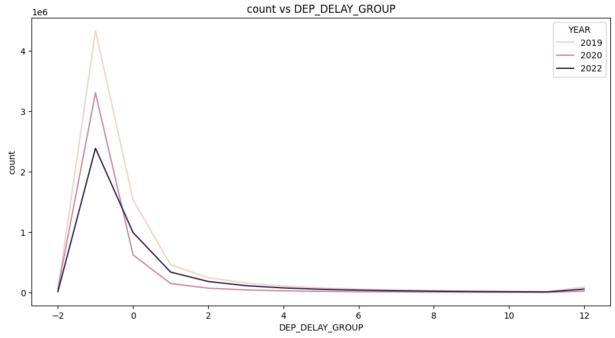
For this phase of the project, in addition to the data we already have, we added the flights from 2022. For our EDA, we will be looking at 2019, 2020, and 2022, so that we can capture the trends pre and post-pandemic. We analyzed the number of flights by month and it looks like the 2022 trend is very similar to that of 2019.



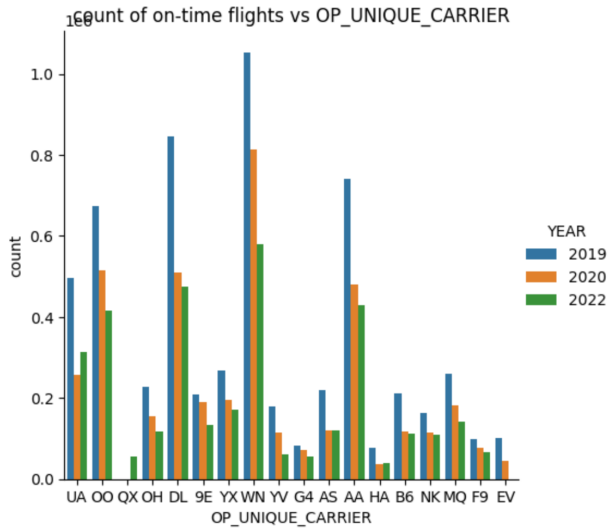
Within those flights, we looked at the ones that were on-time versus delayed. There is still a greater number of on-time flights, but within those that were delayed, 2020 showed a drop in delayed flights, perhaps due to the COVID-19 pandemic, while 2019 and 2022 showed similar proportions.

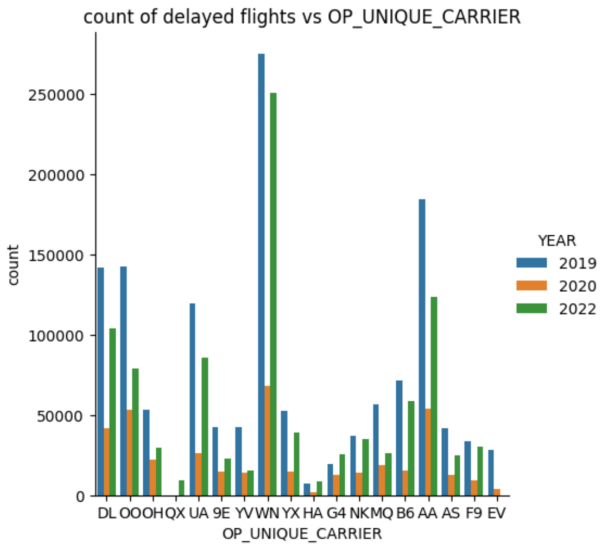


Below is a graph based on the length of the delay. For all three years, it seems like a majority of the flights left either before or on-time.



We then looked into the specific airlines and found that certain carriers had more flights, and in turn generally had more delays. Depending on the airline, the number of on-time versus delayed flights seem to follow the same trend, with the exception of the delayed flights in 2020, which is probably due to the pandemic and reduced number of flights.





Feature Engineering

During the feature engineering phase, we realize we have a need for additional features and a need for feature reduction. We began by performing a PCA on our existing dataset to reduce the dimensionality of the data. Of our original dataset, we retained only eight of the features.

Next, we began exploring additional avenues in terms of features. Engineered features include time-based, graph-based, event-based, and miscellaneous features such as airport maintenance, airline reputations, and natural disasters

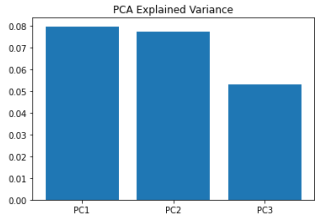
PCA on Dataset Features

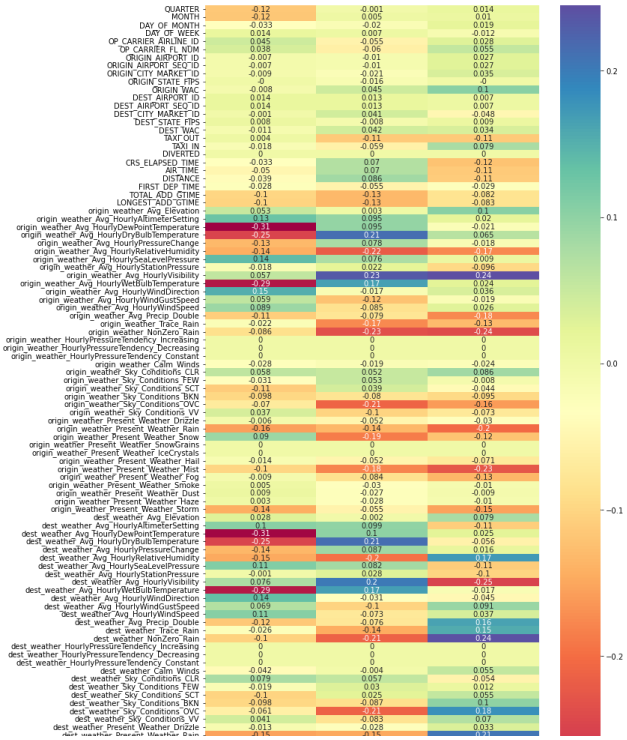
To narrow down features in the original joined dataset, 3 PCAs were performed on numeric columns. Below is a histogram of the PCA explained variances. Explained variance is a “statistical measure of how much variation in a dataset can be attributed to each of the principal components”. All of them are quite low, with a variance below 0.08. PC1 and PC2 have similar explained variances of 0.08 and 0.075 respectively while PC3 is at 0.05. This means that there is not much variation attributed to the principal components.

In heatmap of PCA scores, blue and red scores are the larger magnitude coefficients, and are more important in calculating the component. we have selected 17 variables with absolute value scores above 0.2. A correlation heatmap is plotted to remove intercorrelated features. Thus, our final features from the original dataset to 8 features:

- 'dest_weather_Avg_HourlyDryBulbTemperature'
- 'dest_weather_Avg_HourlyRelativeHumidity'
- 'dest_weather_Avg_HourlyVisibility'
- 'dest_weather_Sky_Conditions_OVC'
- 'origin_weather_Avg_HourlyDryBulbTemperature'
- 'origin_weather_Avg_HourlyRelativeHumidity'
- 'origin_weather_Avg_HourlyVisibility'
- 'origin_weather_Sky_Conditions_OVC'

Below are the functions we used to run PCA and display visualizations.





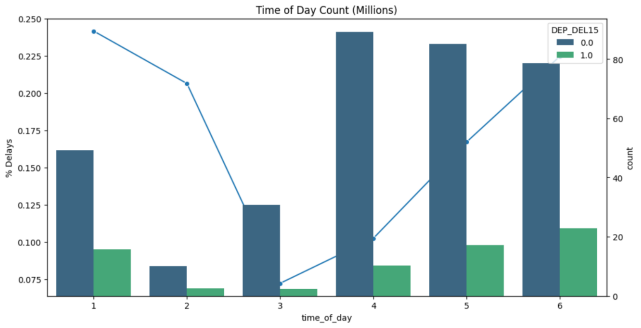
```
def run_PCA(df, cols):     new_df = df.select(cols)     df1 = new_df.na.drop("an ...
```

Show cell

Time-based Features: Time of Day

The first feature we implemented was a time-based feature focused on identifying flight patterns and when flights are flown most often. The flights are broken out by 4hour increments. For example, category one captures flights departing between 12:00AM - 4:00AM, category two is from 4:00AM - 8:00AM etc. We graphed this out in the chart below. This is further broken out to a binary bar chart, separating delayed and non-delayed flights. The line in the chart below helps us determine the delayed flights as percentage of total flights in each category.

As we can see, delayed flights tend to spike up more during the late evening and night, as a prcentage of flown flights. We can interpret that as the airport potentially having fewer resources or perhaps work is not as efficient during those hours, resulting in a delay. This feature may prove to be useful when determining flight delay patterns




```
def add_timeofday(df):     def categorize(date_time): #          formatted_time = ...
```

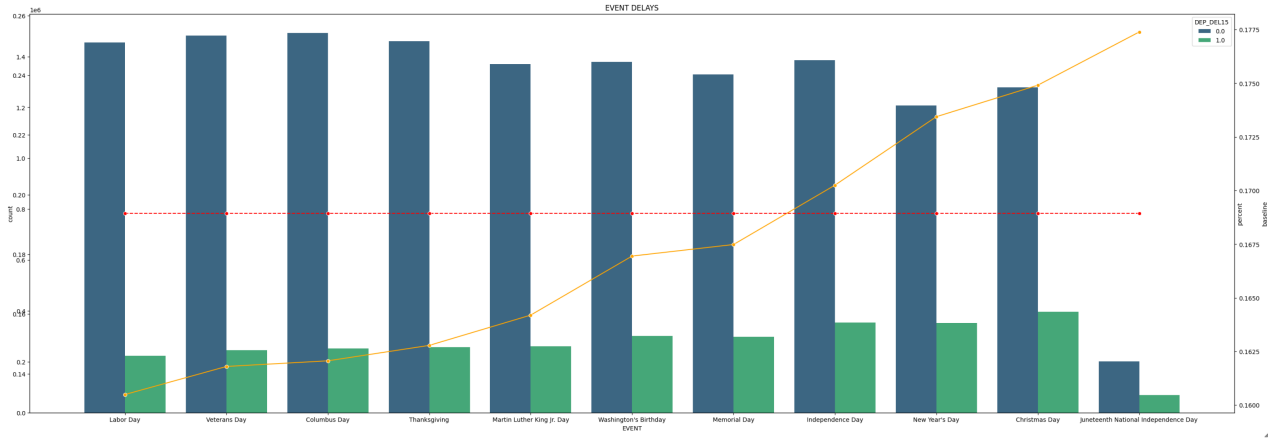
Show cell

National Events

Many passengers opt to fly around holidays for a variety of reasons. Some passengers take advantage of a longer weekend while others fly during these times due to special deals that may be available for flights. Regardless, the holidays are a busy time for all, particularly around Thanksgiving, Christmas, and New Years. This feature was created to classify dates around that period. Due to the uptick in flights around the busier times of year, delays may be expected to occur more frequently.

In the feature, we identify flights that are flown in a plus and minus seven day window to a holiday. We opted for a one week buffer because passengers tend to fly to their destinations around a week in advanced at times and when they depart, they may also follow a similar pattern.

Below is a diagram describing the flights around these holidays. The blue and green represent the number of non-delayed versus delayed flights respectively. The yellow line represents the percentage of delayed flights per holiday and the red line represents the percentage of delayed flights, on average, for non-holiday related flights. As we can see, certain holidays have a much higher delay percentage on average. That being said, the events make up only a fraction of the year as a whole.



add_events(df): returns sql.dataframeb

```
def add_events(df):     from datetime import date     import holidays     us_ho ...
```

Show cell

Graph-based Pagerank Airport

The graph-based feature is the PageRank of airports using the inbound and outbound airports in our joined dataset. The pagerank algorithm computes a rank based on the airport's inbound and outbound flights. Since airports with many connections are more likely to be delayed and will propagate to other airports, pagerank can help our model take this into account.

The correlation between pagerank and DEP_DEL15 (our label) was 0.034, which is not very high. However, our logistic regression show that pagerank has a relatively high feature importance and coefficient.

add_pagerank(df): returns sql.dataframe

```
def initGraph(dataRDD):     """     Spark job to read in the raw data and initia ...
```

Show cell

pagerank_impact

```
def pagerank_impact(df):
    corr = df_pagerank.corr('DEP_DEL15', 'pagerank')
    print(f"The correlation between DEP_DELAY and PageRank is {corr}")
    return corr
```

Natural Disaster

Flights can get delayed due to natural disasters in the origin or destination. Natural disasters can be detected when there are abnormal weather anomalies. Therefore, the column 'natural_disaster' is 1 when the origin or destination average hourly values are 3 standard deviations away from the mean. The standard deviation multiplier is determined by iterating the multiplier to achieve the highest correlation with the target variable.

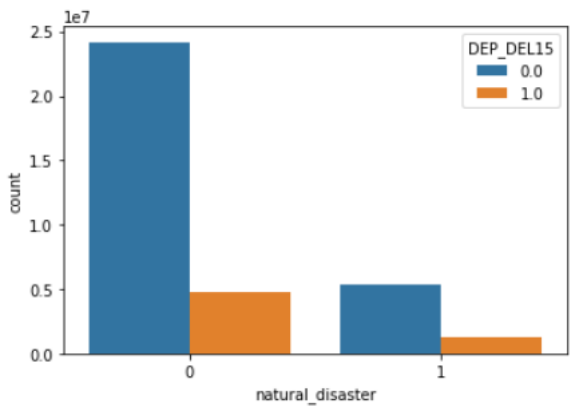
add_natural_disaster(df, multiplier=3.0): returns sql.dataframe

```
def add_natural_disaster(df, multiplier = 3.0):     """     IN: sql.dataframe wi ...
```

Show cell

Natural Disaster EDA

```
def natural_disaster_impact(df, corr=True):
    if corr:
        corr = df.corr('DEP_DEL15', 'natural_disaster')
        print("corr", corr)
    new_grouped = df.groupby('DEP_DEL15', 'natural_disaster').count().toPandas()
    sns.barplot(data=new_grouped, x='natural_disaster', y='count', hue='DEP_DEL15')
    return corr
```



Airline Reputation

The airpline reputation feature was created using a rolling window of the previous year's airline reputation for having delayed flights. For instance, if the flight date was 11-30-2021, the reputation would be calculated from 11-30-2020 to 11-29-2021. If an airline has a trailing reputation for delayed flights, we may be able to assume the same for the following year.

add_reputation(df): returns sql.dataframe

```
# helper functions for calculating year-long interval def calc_interval_first(d) ...
```

Show cell

Airport Maintenance

For Airport maintenance, we used a lagging indicator to determine the overall maintenance delays an airport typically faces. For example, if there were one hundred flights in 2015, and 25% of those flights were delayed due to maintenance, they would have a "reputation" for being delayed 25% of the time in 2016. The percentage or probability of being delayed due to maintenance will only be carried up to one year. E.g. 2018 will only influence 2019 and 2019 would only influence 2020.

add_maintenance(df): returns sql.dataframe

```
def add_maintenance(df):          def next_year(year):         return int(year)+ ...
```

Show cell

Combine all the features

In this step, we take the time to run and combine all of our data. This includes cleaning the dataset as well as adding the above mentioned features into the dataset.

Clean Data

```
def clean_data(df):     df = df.withColumn('YEAR', F.col('YEAR').cast('int')) \  ...
```

Show cell

```
def clean_2014():     # TO BE REFINED #     df_2014_new = (df_2014.withColumn('F ...
```

Show cell

```
##### JIMI'S DATAFRAME ####
data_BASE_DIR = "dbfs:/mnt/mids-w261-joined"
mids_w261_joined = spark.read.parquet(f"{data_BASE_DIR}")
df1 = mids_w261_joined.filter(col('CANCELLED') == 0.0)
df2 = add_natural_disaster(df1)
df3 = add_timeofday(df2)
df4 = add_2014(df3)
df5 = add_pagerank(df4)
df6 = add_reputation(df5)
df7 = add_maintenance(df6)
df8 = add_events(df7)
df9 = clean_data(df8)
df9.write.mode('overwrite').parquet(f"{blob_url}/df_full_14_21")
```

Final Features

In our final dataset, we feature engineered six new features while reducing the original dataset down to eight features. The final feature set contains a total of 15 features, with 9 weather features, 1 graph-based feature, 2 time-based features, and 3 flight features.

Family	Count	List of features	Description
Baseline (PCA-based, weather features)	9	dest_weather_Avg_HourlyDryBulbTemperature'	Destination of Flight Dry bulb temperature by hour
		dest_weather_Avg_HourlyRelativeHumidity'	Destination of Flight Relative Humidity temperature by hour
		dest_weather_Avg_HourlyVisibility'	Desitination of Flight distance of horizontal distance visibility
		dest_weather_Sky_Conditions_OVC'	Destination sky condition amount of cloud cover
		origin_weather_Avg_HourlyDryBulbTemperature'	Origin of Flight Dry bulb temperature by hour
		origin_weather_Avg_HourlyRelativeHumidity'	Origin of Flight Relative Humidity temperature by hour
		origin_weather_Avg_HourlyVisibility'	Desitination of Flight distance of horizontal distance visibility
		origin_weather_Sky_Conditions_OVC'	Origin sky condition amount of cloud cover
		Natural Disasters	Bool if natural disaster occuring
Graph-based	1	Pagerank	Page Rank of airport
Time-based	2	National Holidays	Bool if corresponding national holiday occuring
		Time of Day	Time of Day broken down into 4 hr increments
Flights Features	3	DEP_DEL15'	bool if departure delay of >= 15 min
		Airport Maintenance	percent of flights delayed more than 45 min over total flights delayed for the past year
		Airline Reputation	percent of flights delayed flights over past year (rolling window)

Data Pipeline Discussion

Pipeline Run Time

Our pipeline run time was initially pretty slow at around a 10-15 minutes for each pipeline. This improved significantly with function implementation and caching, and now averaging a few minutes.

Leakage

Leakage is when information should not be known to the model at the time of prediction is fed to the model. An example is providing the weather data that occurs at the same time of the flight. But we want to predict the delay 2 hours prior to allow the customer time to react to the delay. Thus, The model should only have access to the weather data 2 hours before the flight when making a prediction for that flight. To prevent leakage or violating cardinal sins of ML, we ensure that we split and set aside data for train, validation, and test. 2021 year was only used for blind test evaluation at the end of the process. We avoided EDA on 2021 as much as possible due to being a blind test set. It was never used during the training process. We looked for any dependencies. All features were offset by minimum 2 hours if not more to ensure no data leakage to the model.

```
df_full_14_21 = spark.read.parquet(f"{blob_url}/df_full_14_21/") omit_columns =  ...
```

Show cell

```
from pyspark.ml.feature import OneHotEncoder,StringIndexer from pyspark.sql.type ...
```

Show cell

Novel Direction - SMOTE

When the team ran the initial baseline model, the train F1 score was 0.52 whereas the test score was 0.28. We believe the Logistic Regression model we built was fine and the features we selected were presumed to have some predictive power. For a while, we wondered how we could improve the model. We went back to the dataset to perform some additional EDA to see if there was anything we might have been missing. After some time, we came to the realization that the label classification was imbalanced! The number of non-delayed flights outnumbered the delayed flights by a ratio of nearly 5:1.

To handle the class imbalance, we decided to implement a SMOTE technique where we would create synthetic data. This would bring the two classes closer to a balance with a final ratio of 0.97.

```
df = spark.read.parquet(f"{blob_url}/df_model").filter(F.col('YEAR') < 2021)  de ...
```

Show cell

Early Stop

For the data we use for early stopping, we used the F1 score, an ML metric, of the model evaluated on a validation set of data. Due to flunctuations between F1 scores during training, we persevere for 3 epochs/ iterations to ensure consistent convergence. Early stopping helps to prevent underfitting/ overfitting the model on our metric of interest in an optimal manner. With early stopping, we can allow the model to train on many more epochs without having to manually test different number of iterations and thereby save time. This method ensures the model has trained enough to not underfit by ensuring at least a certain number of consistent validation scores. Conversely, the model is not overfitting to the train set by setting too many iterations beyond the first sign of convergence.

Multilayer perceptron in pyspark does not have a way to implement early stop. We created a custom function iterating the fitting for every 3 iterations by feeding the previous model weights. We split and held validation set to evaluate as our metric for early stop after every 3 iterations. If there are 3 f1 validation scores within a set tolerance of each other, the model is considered converged. Cross validation was implemented, but for this cross validation we treated all the data as independent compared to our prior approach of an expanding window due to time series dependency. This is for feasiblity and to see comparison of the effect on f1 score compared to the expanding window cross validation.

```
def MNN_e(trainDF, features, maxIter=[50], layers=[], blockSize=[128], stepSize= ...
```

Show cell

```
def early_stop(): #trainDF_smote_full = spark.read.parquet(f"{blob_url}/df_model ...
```

Show cell

```
early_stop()
```

```
Dropping 93833 rows...35436418 remaining.
Undersampling trainDF...
major to minor class ratio: 4.872839184945185
Complete.
Completed pipeline set up.
MLP Architecture:
 MLP-33input-2softmax
3th iteration
        new train (f1 validation) score: 0.6146004629006315
6th iteration
        new train (f1 validation) score: 0.6184627799140598
9th iteration
        new train (f1 validation) score: 0.6187965964792662
        previous train score: 0.6184627799140598
        score difference in last 2 iterations: 0.0003338165652063685
12th iteration
        new train (f1 validation) score: 0.6186520451573732
        previous train score: 0.6187965964792662
```

```
        score difference in last 2 iterations: 0.00014455132189306052
        no improvements in the last 3 iterations, returning model with new weights and score
completed 1 fold...
```

Algorithms

In this project, we tried several different models. We tried logistic regression, decision tree, and multilayer perceptron classifier. We used logistic regression as our baseline since it was our most simplistic model. Each were fed the same dataset after vectorizing, standardizing, and one-hot encoding. We ran experiments testing each model with same baseline features that were filtered based on impact from the PCA results. Afterwards, we compared each on the test and training scores to determine the best model classifier on the baseline dataset. Next using the best model classifier we identified, we experimented with different combination of newly engineered features to determine impact of each feature. Further experiments were done with grid search to determine the optimal hyper parameters.

Preprocessing

In the processing step, we will select the feature we want to run on our model. Since the features we used only had about 3000 null values our of and drop any rows with null values. The training dataset consists of data from 2015 to 2020 (inclusive) and the test dataset consists of data in 2021. To address the severe class imbalance in the training dataset, the majority class (not delayed flights) is undersampled down to about 20% of the data, which is equivalent to the ratio between major and minor class. Although most of the experiments are run using this preprocessing pipeline, we will also test the SMOTE strategy, which we will discuss in later stages.

1. Select Features
2. Drop NA's
3. Sampling on TrainDF

```
def get_train_test(df_full, features, label, year, undersample=False):     """ S ...
```

Show cell

Modeling Pipeline Stages

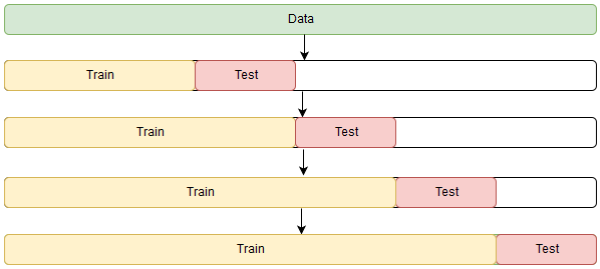
Our pipeline model consists of a vector assembler, standard scaler, and MLlib model. This pipeline allows us to vectorize and normalize our features before fitting to our model of choice.

1. Vector Assembler
2. Standard Scaler
3. MLlib Model (Logistic Regression, Decision Tree, Multilayer Perceptron Classifier)

```
def build_pipeline(modelType, features):
    # Vectorize
    assembler = VectorAssembler(inputCols=features, outputCol="features")
    # Normalize data
    scaler = StandardScaler(inputCol="features", outputCol="features_scaled", withMean=True)
    if modelType == 'lr':
        model = LogisticRegression(labelCol="label", featuresCol="features_scaled")
        pipeline = Pipeline(stages=[assembler, scaler, model])
    elif modelType == 'mlp':
        model = MultilayerPerceptronClassifier(labelCol='label', featuresCol='features_scaled')
        pipeline = Pipeline(stages=[assembler, scaler, model])
    elif modelType == 'dt':
        # Decision tree does not need to scale data
        model = DecisionTreeClassifier(labelCol="label", featuresCol="features")
        pipeline = Pipeline(stages=[assembler, model])
    else:
        print("Please specify modelType as lr, mlp, and dt")
    print('Completed pipeline set up.')
    return pipeline
```

Custom Cross Validator

Flight and weather data is a time series data, so we have implemented 5-fold expanding window cross validation by year. In the first fold, 2015 data is the train set and 2016 is the test set. In the second fold, 2015 and 2016 data were the train set and 2017 data is the test set. We continue this train/test split until we reach our last year of 2021 as the test set, and the years 2015-2020 as the train set.




```
def custom_d(trainDF):
    d = {}
    i = 1
    for year in range(2015, 2020):
        key = 'df' + str(i)
        d[key] = trainDF.filter(trainDF.YEAR <= year + 1)\
                        .withColumn('cv', F.when(trainDF.YEAR <= year, 'train')
                                           .otherwise('test'))
        i += 1
    return d
```

MLlib Models

Our prediction model aims to predict whether or not a flight will be delayed for more than 15 minutes, 2 hours in advance. Our target variable is 'DEP_DEL15' from the flights dataset, which is a binary variable with 1 for more than 15 minutes delay and 0 for less than 15 minutes delay. To solve this binary classification problem, we will be using Logistic Regression as our baseline and Multilayer Perceptron Classifier to push performance. All of our models follow the same pre-processing and pipeline model function, with minimal adjustments to function calls according to object type.

Logistic Regression

Our baseline model is logistic regression. It is a linear method with the logistic loss function:

$$L(w; x, y) := \log(1 + \exp(-yw^T x))$$

where w is the weight matrix, and x is a new datapoint. The model makes a prediction by applying the logistic function:

$$f(z) = \frac{1}{1 + e^{-z}}$$

where $z = w^T x$.

Source: spark.apache.org (https://spark.apache.org/docs/3.3.1/ml-classification-regression.html)

Decision Tree

In Phase 3, 21 experiments were run in total, starting with the 17 baseline features produced from principal component analysis results. Then, graph-based and time-based features were added in a stepwise fashion to see their impact on the model. Since PCA only accounted for numerical columns, other categorical variables from the flights dataset (unique carrier, origin, and destination) were added one-by-one.

The flights categorical variables and time-of-day time-based feature did not improve the model. Our final model includes the 17 baseline weather features in addition to pagerank of airports, holiday_bool, and leading_flights_count.

Decision trees is a popular method for classification and regression machine learning tasks because they were easy to interpret and can handle categorical features and non-linear feature interactions. It is a greedy algorithm that performs a recursive binary partitioning of the feature space. The splits are chosen to maximize the information gain at a tree node and node impurity (Gini impurity for our case) is a measure of homogeneity of the labels at the node.

$$\text{IG}(D, s) = \text{Impurity}(D) - \frac{N_{left}}{N}\text{Impurity}(D_{left}) - \frac{N_{right}}{N}\text{Impurity}(D_{right})$$

$$\text{Gini}(D) = 1 - \sum_{i=1}^{k} {p_i}^2$$

D is dataset that contains samples from k classes. Probability of samples belonging to class i at a given node denoted as pi

Source: spark.apache.org (https://spark.apache.org/docs/latest/mllib-decision-tree.html)

Multilayer Perceptron Classifier

Multilayer perceptron classifier (MLPC) is a classifier based on the feedforward artificial neural network (https://en.wikipedia.org/wiki/Feedforward_neural_network). MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node's weights w and bias b and applying an activation function. This can be written in matrix form for MLPC with $K + 1$ layers as follows:

$$y(x) = f_K(...f_2(w_2^T f_1(w_1^T x + b_1) + b_2)... + b_K)$$

Nodes in intermediate layers use sigmoid (logistic) function:

$$f(z_i) = \frac{1}{1 + e^{-z_i}}$$

Nodes in the output layer use softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$$

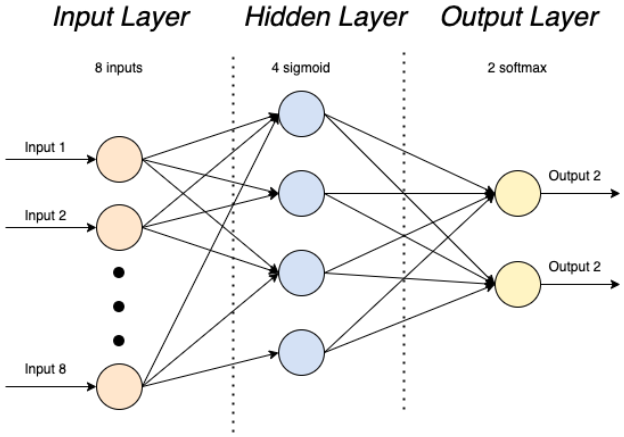
The number of nodes N in the output layer corresponds to the number of classes.

MLPC employs backpropagation for learning the model. We use the logistic loss function for optimization and L-BFGS as an optimization routine.

Source: spark.apache.org (https://spark.apache.org/docs/3.3.1/ml-classification-regression.html#multilayer-perceptron-classifier)

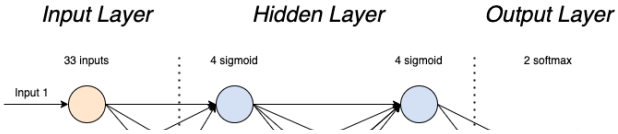
MLP Architecture 1

Our first architecture for the MLP model has 8 features selected using PCA as the input, a single hidden layer with 4 neurons, and an output softmax layer with 2 neurons for the binary class. Most of the experiments are done using this architecture in order to benchmark and compare the effects of adjusting parameters and features.



MLP Architecture 2

Our first architecture for the MLP model has 8 features selected using PCA as the input, a two hidden layer with 4 neurons, and an output softmax layer with 2 neurons for the binary class.



Grid Search Implementation

For MNN, we customized the grid search with parameters for maxIter, stepSize, and solver. We experimented step sizes 0.1 and 0.03 (default). Since step size of 0.03 produced better results by only a small margin (1e-4 for scores) at the cost of more time, we opted for 0.1 step sizes for quick experiments. We also tried different solvers, and found that l-bfgs produced the best result. Moreover, we looked at increasing the number of iterations, but since the results stayed the same, we kept it at 20 iterations for most trials. Lastly, we looked at different number of neurons and layers and will discuss further observations in the results.

Logistic Regression

```
def LR(trainDF, features, maxIter=[15], regParam=[0.0], elasticNetParam=[0.0]):  ...
```

Show cell

Decision Tree

```
def DT(trainDF, features):     """     IN: pyspark.sql.DataFrame, list of featur ...
```

Show cell

Multilayer Neural Network

```
def MNN(trainDF, features, maxIter=[50], layers=[], blockSize=[128], stepSize=[0 ...
```

Show cell

Evaluation Metrics

F1 score was our method of choice due to flight prediction delay set being inherently imbalanced with many more non-delayed flights compared to delay flights. As a harmonic mean of precision and recall, F1 score helps us detect a situation where the model predicts a singular value for every situation due to uneven distribution of classes. F1 score is also a singular number with simple interpretation that enables ease of comparison. We weighed precision and recall equally due to our concern of overpredicting only a single class for all values. We believe customers value being able to reliably know whether their flight is delayed ahead of time increases customer satisfaction and thereby retain their business. In this business case, we hope to use customer life time value and customer retention percentage to monitor the performance of our model in a business setting.

Model performance is derived from the matrix of true positive (TP), true negative (TN), false positive (FP), and false negative (FN). Metrics are derived from this matrix and helps to place importance on one or more of these counts.

Real \ Prediction	positive	negative
positive	TP	FN
negative	FP	TN

Both F1 score and balanced accuracy needs to be considered because F1 score doesn't care about how many true negatives are being classified while balanced accuracy does not consider true positives.

F1-score balances precision and recall.

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

In F1, **Recall**, also known as sensitivity, is the number of true positives out of true positives and false negatives $(\text{Recall} = TP/(TP + FN))$. **Precision** is the number of true positives out of all the true positives and false positives $(\text{Precision} = TP/(TP + FP))$.

On the other hand, **Balanced accuracy** is a mean of recall and specificity.

$$\text{Balanced Accuracy} = \frac{\text{Recall} + \text{specificity}}{2}$$

In balanced accuracy, specificity is the true negative rate $(\text{Specificity} = TN/(TN + FP))$.

Source: neptune.ai (https://neptune.ai/blog/balanced-accuracy#:~:text=F1%2Dscore,accuracy%20of%20an%20individual%20test.)

MLPC - evaluation functions

Show code

Training

```
def training_summary(pipeline_model, params=False):     training_summary = pipel ...
```

Show cell

Evaluate

```
def evaluate_dt(pipeline_model, features, testDF, trainDF):
    # about the model
    model = pipeline_model.stages[-1]
    print(f"Feature Importance: ")
    feat_dict= {}
    for col, val in sorted(zip(features, model.featureImportances),key=lambda x:x[1],reverse=True):
      feat_dict[col]=val
    feat_df = pd.DataFrame({'Feature':feat_dict.keys(),'Importance':feat_dict.values()})
    values = feat_df.Importance
    idx = feat_df.Feature
    plt.figure(figsize=(10,8))
    clrs = ['green' if (x < values.max()) else 'red' for x in values ]
    sns.barplot(y=idx,x=values,palette=clrs).set(title='Important features')
    plt.show()

    print("TRAIN DF Metrics")
    predictions = pipeline_model.transform(trainDF)
    # compute TN, TP, FN, and FP
    # Calculate the elements of the confusion matrix
    TN = predictions.filter('prediction = 0 AND label = prediction').count()
    TP = predictions.filter('prediction = 1 AND label = prediction').count()
    FN = predictions.filter('prediction = 0 AND label <> prediction').count()
    FP = predictions.filter('prediction = 1 AND label <> prediction').count()
    # calculate accuracy, precision, recall, and F1-score
    accuracy = (TN + TP) / (TN + TP + FN + FP)
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    f1 = 2 * precision * recall / (precision + recall)
    balancedAccuracy = (precision + 1 - recall)/2
    # print metrics
    print('n precision: %0.3f' % precision)
    print('n recall: %0.3f' % recall)
    print('n accuracy: %0.3f' % accuracy)
    print('n f1 score: %0.3f' % f1)
    print('n balancedAccuracy: %0.3f' % balancedAccuracy)
    bcEvaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")
    # Evaluate the model's performance based on area under the ROC curve and accuracy
    print(f"Area under ROC curve: {bcEvaluator.evaluate(predictions)}")
    train_metrics =
MulticlassClassificationEvaluator(metricName="weightedFMeasure").evaluate(cvModel1.bestModel.transform(trainDF))
    print("weightedFMeasure: ", train_metrics)

    print("TEST DF Metrics")
    predictions = pipeline_model.transform(testDF)
    # compute TN, TP, FN, and FP
    # Calculate the elements of the confusion matrix
    TN = predictions.filter('prediction = 0 AND label = prediction').count()
    TP = predictions.filter('prediction = 1 AND label = prediction').count()
    FN = predictions.filter('prediction = 0 AND label <> prediction').count()
    FP = predictions.filter('prediction = 1 AND label <> prediction').count()
    # calculate accuracy, precision, recall, and F1-score
    accuracy = (TN + TP) / (TN + TP + FN + FP)
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    f1 = 2 * precision * recall / (precision + recall)
    balancedAccuracy = (precision + 1 - recall)/2
    # print metrics
    print('n precision: %0.3f' % precision)
    print('n recall: %0.3f' % recall)
    print('n accuracy: %0.3f' % accuracy)
    print('n f1 score: %0.3f' % f1)
    print('n balancedAccuracy: %0.3f' % balancedAccuracy)
    bcEvaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")
    # Evaluate the model's performance based on area under the ROC curve and accuracy
    print(f"Area under ROC curve: {bcEvaluator.evaluate(predictions)}")
    bcEvaluator = MulticlassClassificationEvaluator(metricName="weightedFMeasure")
    # Evaluate the model's performance based on area under the ROC curve and accuracy
    print(f"weightedFMeasure: {bcEvaluator.evaluate(test_metrics)}")
```

```
    return predictions
```

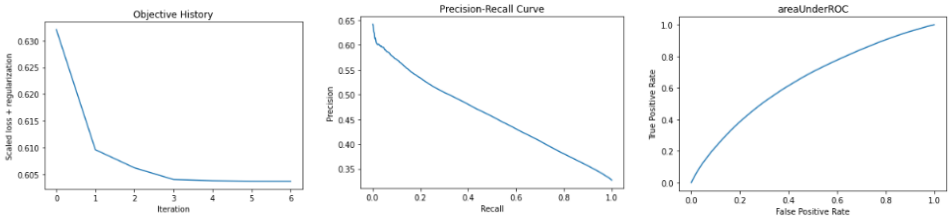
MLP SAVE METRICS

Show code

Experiments, Results and Discussion

Logistic Regression Results

Majority of our logistic regression experiments were conducted in Phase 3 during which we found that L1/L2 regularization parameters did not improve performance, undersampling improved skewed F1 scores, and less than 10 iterations were required for the model to converge. Once we have established our baseline, we opted to focus our efforts on the multilayer perceptron classifier in order to benchmark feature impact while fine-tuning the neural network. Below is our Phase 3 results that compare metrics with and without sampling. Without sampling, the average F1 score was 0.900 due to the fact that the major to minor class ratio was 4.87 and the model merely predicted all flights as on-time. With sampling, the average F1 score dropped to 0.523, which appears more reasonable given the AU-ROC and loss curves for the best model. In the interest of conciseness, experiments with deprecated features were omitted. Instead, we obtain a new best logistic regression model using PCA features, events, airport pagerank, airline reutation, airport maintenance percentage, and time of day to compare with the MLPC final model. As you can see in the training curves below, the model quickly converted within 7 iterations. This gave us an indication to lower the maxIteration for our MLPC model as well. Unfortunately, the PR curve did not appear to have good results and AU-ROC was 0.6444.



Phase 3 Experiments

Show code

Table

	filepath	cluster size	training time (min)	regParam	elasticNetParam	A
1	lr-regParam0-elasticNetParam0-noUndersampling	10	4.4	0	0	0
2	lr-regParam0-elasticNetParam0-withUndersampling	10	4.67	0	0	0

Showing all 2 rows.

Best Logistic Regression Model (Phase 4)

Show code

Table

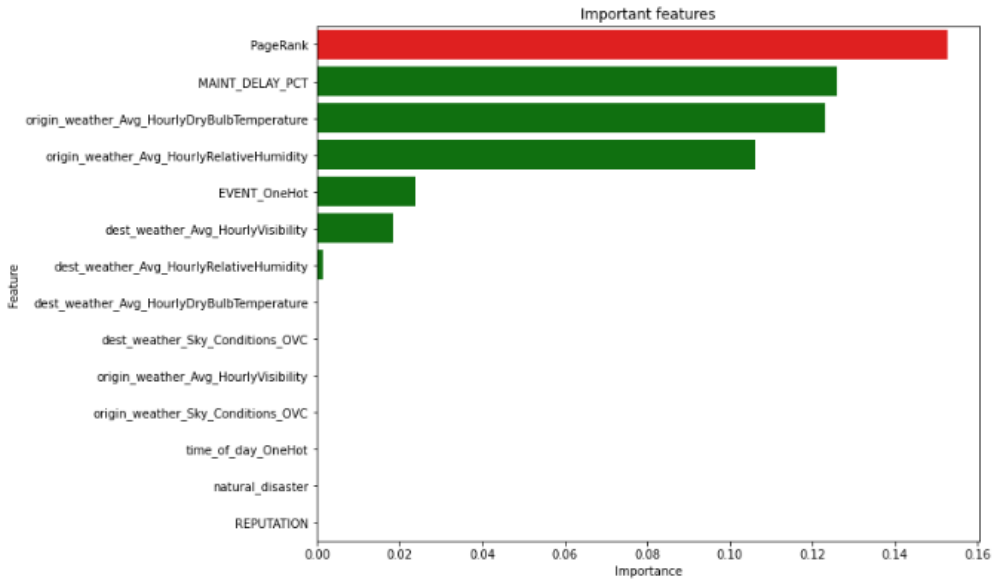
	CV Best Model	Architecture	Test weightedF1	Test Balanced Accuracy	Train weightedF1
1	lr_4	LR-0.0regParam-0.0elasticNetParam	0.7641114456761416	0.8178400082244285	0.60836195825838

Showing 1 row.

Decision Tree Results

Much like logistic regression, our decision tree model experiments with hyperparameter tuning were mainly conducted in Phase 3, which we observed that the tree was very shallow despite added features and parameters such as minWeightFractionPerNode and maxDepth did not produce any significant improvements to the model. In Phase 4, we ran a decision tree model with PCA-selected features and new engineered features to compare with our MLPC best model. This includes PCA features, events, airport pagerank, airline reutation, airport maintenance percentage, and time of day. Our best model obtained a test weighted F1 score of 0.7506, test balanced accuracy of 0.702, train weighted F1 score of 0.3637, and train balanced accuracy of 0.833. It took 2.1 minutes and 10 clusters to run the model. Below is a bar graph of feature importance derived from the decision tree model. PageRank, Airport Maintenance Percentage, and Events were all new features that were among the top features. 3 PCA features were also included in the top features. The rest of the features were not used and indicates that future work should involve increasing the correlation of these features by adjusting windows and feature calculations.

```
Feature Importance:
```



Decision Tree Experiments (Phase 3)

Show code

Table

	file	cluster_size	training_time	maxDepth	minWeightFractionPe
1	dt_1_0.0_joined_df_baseline	10	11.17	1	0
2	dt_1_0.0_joined_df_new_pagerank	10	7.36	1	0
3	df_joined_pagerank_leadingflights	10	9.49	1	0
4	df_joined_pagerank_holidaybool	10	8.37	1	0
5	df_joined_pagerank_holidaybool_leadingflights	10	4.86	1	0
6	df_joined_pagerank_holidaybool_leadingflights_timeOfDay	10	4.89	1	0

Showing all 20 rows.

Best Decision Tree Model (Phase 4)

Show code

Table

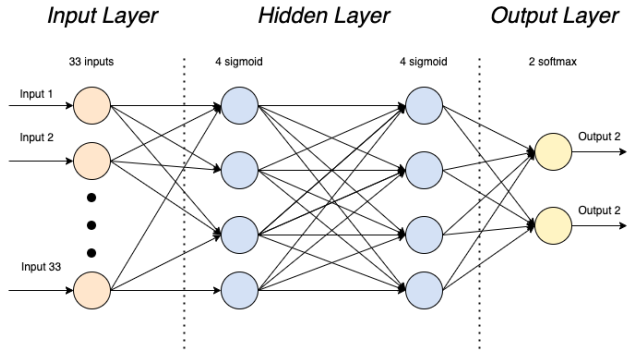
	CV Best Model	Architecture	Test weightedF1	Test Balanced Accuracy	Train weightedF1	Train Balanced Ac
1	dt_1	dt_1	0.7505796667121961	0.702	0.3636703685761434	0.833

Showing 1 row.

Multilayer Perceptron Classifier Results

Best model

Our baseline multilayer perceptron model consists of 8 inputs (numerical features from PCA), one 4-neuron sigmoid hidden layer, and one 2-neuron softmax output layer. It produced a test weighted F1 score of 0.7494, a test balanced accuracy of 0.5012, a train F1 score of 0.5534, and a train balanced accuracy of 0.5062. In contrast, our best model has the architecture of 33 inputs, two 4-neuron sigmoid hidden layers, and 2-neuron softmax output layer. The model obtained a test weighted F1 score of 0.7651, a test balanced accuracy of 0.5259, a train F1 score of 0.6149, and a train balanced accuracy of 0.5450.



Tuning

Our experiments involve train set sampling, feature engineering, and changing the number of layers and neurons, and hyperparameter tuning max iterations, step size, and solver. Continuing from phase 3, we conducted a simple minor-to-major class ratio undersampling on the major class (on-time flights) to address class imbalance. Without sampling, the logistic regression model in phase 3 predicted all flights to be (on-time) to boost metrics. With sampling, the model is better able to optimize without the interference of class imbalance. Now in phase 4, we attempted to apply SMOTE. The train weighted F1 increased to 0.676 and train balanced accuracy increased to 0.605, which is approximately a 0.05 increase from the baseline for both scores. Unfortunately, we were not able to obtain test weighted F1 and balanced accuracy scores due to issues with aligning our test and train set within the time constraint. Future steps would be to debug our SMOTE functionality and test its impact on the test set.

Feature engineering contributed the most to our score increases, we ran experiments with the baseline PCA features and a single new engineered feature. Most notably, REPUTATION increased test weighted F1 score from out baseline 0.749 to 0.765 and test balanced accuracy from 0.501 to 0.526. Train scores increased more dramatically, with train weighted F1 score increasing from 0.553 to 0.615 and train balanced score from 0.506 to 0.545. Feature engineering's the large contribution to performance improvement and the test scores being higher than the training scores leads us to hypothesize that the model is underfitting. This means that the model is not complex enough and the addition of features adds to model complexity. Thus, much of our experiments with hyperparameters, neurons, and layers aim to increase the complexity of the model. Yet, these adjustments did not make significant impact compared to the effect of feature engineering.

Grid search selected the l-bgs as the better model over gd, holding all other parameters constant. Increasing the step size from 0.01 and decreasing the number of maximum iterations from 40 to 15 did not significantly change train and test scores. This means that the model is stable and has reached convergence at 15 iterations. Using our final set of features, we first experimented with increasing the number of neurons in our single layer from 4 to 7. We hypothesized that increasing the number of neurons in response to the increase in number of inputs (the baseline model has 8 inputs while the final model has 33 inputs) will help retain more information. However, the scores only increased in the magnitude of 3e-4, which is not significant enough to call it an improvement (mlp_5 and mlp_6). Experiments 'mlp_7' increased the number of hidden layers to two, with four neurons each. Compared to our benchmark mlp_5 that has one hidden layer of 4 neurons, mlp_7 only increased test weighted F1 by 7e-4 , test balanced accuracy by 2.2e-3, train weighted F1 by 4.7e-3, and train balanced accuracy by 3.5e-4. An additional trial with an additional 4-neuron layer(mlp_14) did not perform better than our single-hidden layer model. Overall, we hypothesize that class

imbalance and feature selection overshadowed the potential of hyperparameter tuning and neural network architecture changes. Nevertheless, there are many non-linear features in our model that may be better-captured using more complex activation functions such as 'relu' as opposed to multilayer perceptron classifier's sigmoid.

```
mlp_experiments = spark.read.parquet(f"{blob_url}/MLPv2/experiments_table")
display(mlp_experiments)
```

Table

	CV Best Model	Architecture	Test weightedF1	Test Balanced Accuracy	Tra
1	mlp_13	MLP-33input-15sigmoid-4sigmoid-2softmax	0.7644869910947227	0.5235915661409185	0.6
2	mlp_14	MLP-33input-4sigmoid-4sigmoid-4sigmoid-2softmax	0.7643711642973161	0.5242713748408898	0.6
3	mlp_4	MLP-8input-4sigmoid-2softmax	0.7493901637283577	0.501159881238278	0.5
4	mlp_7	MLP-33input-4sigmoid-4sigmoid-2softmax	0.7650595284672154	0.5259417646692461	0.6

Showing all 13 rows.

Comparing Models | Successes & Surprises

The best model was the MLPC Neural Network model with 33 inputs, two 4-neuron sigmoid hidden layers, and 2-neuron softmax output layer. The model obtained a test weighted F1 score of 0.7651, a test balanced accuracy of 0.5259, a train F1 score of 0.6149, and a train balanced accuracy of 0.5450. We specified the step size to be 0.1 and max iteration to be 25, leaving the rest as default. Our experiments demonstrated that the MLPC model converged quickly - even 10 iterations produced metrics only lower by a magnitude of 1e-3. The potential for hyperparameter tuning to improve the model was largely affected by class imbalance and underfitting, hinted through low train scores and higher test scores. Number of neuron and layers also did not significantly impact model performance. We hypothesized that some features were non-linear and perhaps different activation functions such as relu would be more effective in improving model complexity and addressing underfitting as future work. Feature engineering made the most contribution. Namely, adding airline reputation boosted train and test scores the most. In comparison, logistic regression and decision tree produced similar results when using the same features as the MLPC neural network best model. Logistic Regression's best model converged in 7 iterations without regularization and produced stable scores - test weighted F1 score of 0.764, a test balanced accuracy of 0.818, train weighted F1 score of 0.608 and train balanced accuracy of 0.682. The best decision tree model produced a test weighted F1 score of 0.751 and a test balanced accuracy of 0.702. Although decision tree produced high test balanced accuracy, train weighted F1 score was low (0.364). A surprising observation was that the decision tree feature importance ranking revealed that the model did not use airline reputation in its tree even though it was one of the most important features in the MLPC model.Smote produced as expected large improvement of 0.676 F1 score on the base MLP model when compared to our custom undersampling method with 0.5 F1 score. Early stopping training average score with an independent k fold cross validation with 0.619 F1 score had roughly the same as our expanding window cross validation score of 0.614 with no early stopping. This is suprising how similar they are even though this dataset has time series dependencies.

Gap Analysis

Majority of the teams on the leaderboard reported F1 scores as their main metric and thus we will be using our F1 score as to benchmark with other teams as well. Scores on the leaderboard ranged from 0.5 to 0.866. Most utilized 6 workers and recent reported training times were approximately 10 minutes, with 10 scores reporting times in the 1-hour range. This is similar to our experiments with 10 clusters and training time of 3 to 7 minutes. In terms of algorithm, many used logistic regression as their baseline with one or more advanced algorithms (random forest, gradient boosted trees, neural network). Some reported "major surprises" matched our observations while others contradicted. Like us, many reported that feature engineering significantly boosted performance and prior flight information were very important features. Unlike us, Team Pi-8's decision tree model took the undersampling approach but faced overfitting while their reported recall of 0.66 was similar to our weighted Recall of 0.68. In light of this finding, we recommend trying lower sampling ratios to address underfitting. Team House Spark stood out as the best-performing group with thorough experiments on multiple algorithms. They tested logistic regression, linear regression, random forest, decision tree, gradient boosted tree, and multilayer perceptron classification model. All of their models achieved approximately a test F1 score of 0.865, with the multilayer perceptron model performing the best with an F1 score of 0.882. This is 0.117 higher than our best model's test weighted F1 score. Much like other comments about past flight information, their top features were percent delays in the past day, percent cancellations in the past day, and count flights in the past day. In comparison, our pagerank was partitioned by year, national events on the entrie dataset, and airport maintenance percent in the past year, natural disaster based on weather data from 2 hours ahead of scheduled flight, and airline reputation in the past day. In reflection, airline reputation had the largest impact on our model performance. Thus our recommended next step is to create more features based on flight information as recent as the past day or even 2 hours prior. To summarize recommended steps derived from gap analysis, we recommend creating more features based on flight information from the past day or hour, such as number of scheduled flights, rate of flight increase in the past hour, and cancellations/delays by airline in the past hour. Then, undersampling ratio and SMOTE can be further refined to help improve training scores. Finally, neural network was the right step forward and utilizing tensorflow and pytorch can help create even more sophisticated models that incorporate early-stopping and callback functionalities that make training more seamless than what is provided in MLlib.

Performance & Scalability

Throughout the project, our team encountered multiple unforeseen scalability concerns. The first issue was writing to blob after joining the three datasets (airlines, weather, and stations) together. We were unsuccessfully in writing to blob when attempting with the entire dataset at once, and so the solution was to write to blob in piecemeal so that the DAG does not get overwhelmed. Another issue was the cleaniness of the data. As the data has millions of records, having a thorough EDA and truly understanding each column's contents were imperative for scalability. With the data given, there were many nulls and formatting issues we needed resolve before scaling up, otherwise we would lose too much data. Lastly, worker resources were a limiting factor in our work. For the first two phases of the project, we were only given 4 workers, which resulted in high run times for all of our work. In phase three, we were given 10 workers for our jobs, but for many of the models, we used all 10 workers almost all the time, and we still encountered delays and aborts in our pipeline when resources were limited. To remediate these delays, we wrote all of our processes into functions and tried to limit the activity on our cluster to one main focus at a time so that we can have more reasonable runtimes.

Limitations & Future Work

We recognize our model is underfitting due to our test scores being higher than our training model scores. When running experiments, the largest impact to improving model perfomance was due to our feature engineering. Future work to further improve our F1-score would be creating additional features to better capture the variation within flight delays. For future work, we also hope to evaluate other models, the impact of improving some ofour exisitng features to be on a 2 hr rolling window, and other methods implement early stopping.

Some features due to limitations in cluster resources, time, and cluster resources were not able to implement on a rolling window and implement on a 2 hr prior to flight basis even though data leakage issues were addressed so no data less than 2 hour window was fed into the model. Pyspark was also limited in the available packages such as XGB model that could integrate with databricks. The MLP model did not have a way to build in early stopping. The pyspark library was also often not very descriptive.

Conclusion

The goal of this project is to create a machine learning model that predicts flight departure delays 2 hours ahead of time as a tool for airlines to improve flight scheduling and delay response. Data exploration revealed that flight delays were small abnormalities in the entire dataset and correlations were muted. Thus, we hypothesize that creating new features targeted towards factors causing delay can amplify correlations and improve model performance. To adjust for the class impalance, we implemented a SMOTE technique to synthesize data. The models we focused on were: logistic regression, decision tree and multilayer perceptron classifier.

The best model was the MLPC Neural Network model with 33 inputs, two 4-neuron sigmoid hidden layers, and 2-neuron softmax output layer with a test weighted F1 score of 0.7651, a test balanced accuracy of 0.5259, a train F1 score of 0.6149, and a train balanced accuracy of 0.5450. We specified the step size to be 0.1 and max iteration to be 25, leaving the rest as default. The MLPC model converged quickly wihtin 10 iterations and additional neurons and layers lowered metrics. We hypothesized that some features were non-linear and perhaps different activation functions such as relu would be more effective in improving model complexity and addressing underfitting as future work. Feature engineering made the most contribution for all three models. Airline reputation was an important feature in MLPC model while the decision tree model did not use the feature at all. Although decision tree produced high test balanced accuracy, train weighted F1 score was low (0.364), confirming the need for both F1 score and balanced accuracy.

Our results were similar to the majority of the project leaderboard with test weighted F1 scores in the range of 0.72 to 0.78 across all models, using 10 clusters and 3~7 for training time. We also faced similar challenges with severe class imbalance still affecting performance after undersampling in addition to feature engineering making the most contribution to model performance. Team House Spark performed the best with an F1 score of 0.882 and their top features and comments suggest that features engineered using past flight information over a small window (hours before the flight) had a positive affect on model performance. Throughout the project, we faced severe class imbalance and scalability challenges when writing our datasets to blob storage and conducting EDA calculations. In the future, we recommend creating more features with narrow time-windows and testing more sampling techniques to address severe class imbalance.