Will Your Flight be Delayed?

MIDS W261: Machine Learning at Scale | UC Berkeley School of Information | Spring 2022 | Final Project Report

Team 12: "CAWVS" | *Sections: 3 and 4*

Stephen **C**hen | Mahesh **A**rumugam | Radia **W**ahab | Jericho **V**illareal | Ramanuj **S**ingh

1. Introduction

Air travel has a significant role in shaping economies of states and thus it is important to optimize every aspect of the flight cycle to increase the quality of airline's services, most especially the consistent on-time departure flights. However, due to multiple variables and external factors, airline flight delay is inevitable, and thus have to be accounted for by the airline company executives in order to optimize value to multiple stakeholders -- from the shareholders to the customers.

Fortunately, since June 2003, airlines that report on-time data also report the causes of delays and cancellations to the Bureau of Transportation Statistics. The airlines report the causes of delay in broad categories created by Air Carrier On-Time Reporting Advisory Committee. Specifically these categories are the following [1]

- Air Carrier: delays due to circumstanes within the airline's control such as maintenance problems, baggage loading, aircraft fueling, etc.
- Extreme Weather: actual or forecasted significant meterorological conditions, such as blizzard or hurricane, that prevents airline operations
- National Aviation System (NAS): delays and cancellations attributable to a broader set of conditions like air traffic control
- Late-arriving aircraft: previous flight with the same aircraft arrived late, causing the present flight to depart late (knock-on effect)
- Security: delays and cancellations caused by evacuation of aircraft or terminal and inoperative screening equipment and/or long lines in excess of 29 minutes at screening areas.

How do airlines think about flight delays?

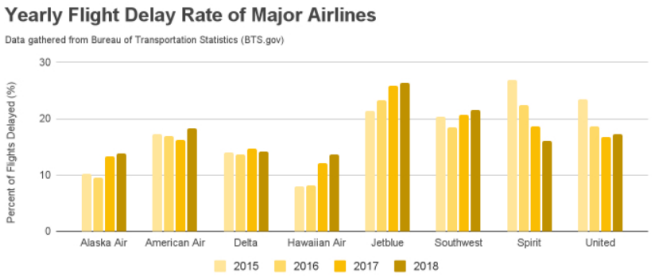
Based on data gathered from the Bureau of Transportation Statistics (BTS.gov) across the major airport hubs in the United States from 2015 to 2018, at least 10% of the flights get delayed on a daily basis, and has seen this rate to increase to about 35% to 40% daily. Some of the major airlines have made strides in reducing their own delayed rate or even keeping it at a steady rate, while others have increases in flight delays which in turn affect the airline company's financials.

References

[1] Bureau of Transportation Statistics

1 in 5 flights get delayed daily

Based on 2015 to 2018 data gathered from the Bureau of Transportation Statistics (BTS.gov).



The business case is that in 2018 the total cost of flight delays has been estimated at $28 billion. In aviation, an airline departure are considered to be "on-time" when they depart within 15 minutes of the scheduled flight. This is the basis of the airline's proposition to its customers and also a huge factor in measuring an airline's On-Time Performance (OTP) metric. OTP is a widely used metric for airlines and is an outward demonstration of reliability which can affect multiple facets of the company like sales, branding, and customer satisfaction. In optimizing OTP metric, some airlines would rather create 'ghost flights' than not have to give up airport slots, which is set of permissions to schedule landing or departure at the airport during specific time period.

$28B
estimated annual cost of flight delays in 2018
* assuming $47 per hour as the average value of a passenger's time (direct cost to airlines and passengers, lost demand, and indirect costs), according to FAA/Nextor

$75M
Oman Air paid a record sum to KLM for a pair of airport slots in 2016

$60M
American Airlines paid to SAS for a pair of slots in 2015

$209M
Continental Airlines to GB Airways for 4 pairs of slots in 2007 ($52.2m per pair)

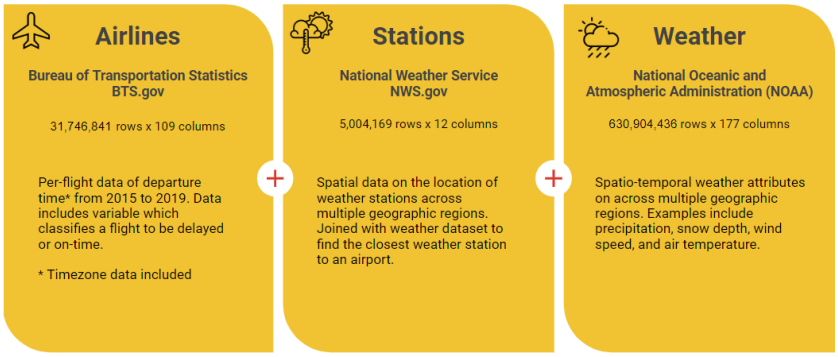
$47M
Delta Airlines paid to unknown airlines for two pairs of slots in 2012

1.1. Datasets

There were three main datasets used in this project.

1. Airlines dataset from the United States Department of Transportation, utilizing the Bureau of Transportation Statistics (BTS.gov)[**1**]. The size of the data set is 31,746,841 rows x 109 columns (documentation). This dataset comprises of per-flight data of departure time* from 2015 to 2019. Data includes variables which classify a flight to be delayed or on-time and other metadata concerning each of the flights.
2. Weather dataset from the National Oceanic and Atmospheric Administration (NOAA)[**2**]. The size of the data set is 630,904,436 rows x 177 columns. This data set comprises Spatio-temporal weather attributes across multiple geographic regions. Examples include precipitation, snow depth, wind speed, and air temperature.

3. Weather station dataset from the National Weather Service NWS.gov. The size of the data set is 5,004,169 rows x 12 columns. This data set comprises spatial data on the location of weather stations across multiple geographic regions. We joined this dataset with a weather dataset to find the closest weather station to an airport.



References

[1] Bureau of Transportation Statistics

[2] Quality Controlled Local Climatological Data (QCLCD) Publication

1.2. Question Formulation

With information from weather and airline data gathered two (2) hours before the planned departure time, can we then build a model that can predict if the flight will be delayed by 15 minutes or more?

We will be looking at airline data, weather station data, and weather data. We will be exploring data transformation techniques and ML pipelines in Spark. We have identified `DEP_DEL15` as the target variable which has a label of value `0` (no delay) or `1` (delay).

This problem is framed as a binary classification problem.

1.3. Evaluation Metrics

The evaluation metrics we utilized were *Recall, Precision, F1 Score, Area under the PR Curve,* and *Area under the ROC Curve.*

- *Recall*: Calculated by $\frac{\text{True Positives}}{\text{True Positives+False Negatives}}$, this metric describes a model with high recall if it can correctly predict most of the positive cases even if it misses some few actual positive cases as negative. A model with low recall is unable to correctly predict the positive cases.
- *Precision*: Calculated by $\frac{\text{True Positives}}{\text{True Positives+False Positives}}$, this metric describes if a model is precise if the positive predictions contain very few false positives or noisy if the positive predictions contain many false positives.
- *F1 Score*: This metric combines both precision and recall metrics. Calculated as the harmonic mean of precision and recall, the F1 score gives equal weight to both precision and recall (i.e., $\frac{2*\text{precision*recall}}{\text{precision+recall}}$). A high F1 score means that both precision and recall are high. The F1 score is an ideal metric for imbalanced data.
- *Area under the ROC Curve*: A receiver operating characteristic curve (ROC curve) is an evaluation tool for binary classification problems. It plots the true positive rate against the false positive rate. The area under the ROC curve represents how well the model can distinguish between classes. An area under the ROC curve closer to 1 means that great at predicting true positives and true negatives. However, this metric may not be the most effective when evaluating on imbalanced data.
- ***Area under the PR Curve***: A precision-recall curve (PR curve) is another evaluation tool for binary classification problems. It plots the precision and recall. The area under the PR curve represents the measure of delayed flights that were correctly classified out of all of the delayed flights. This metric would be ideal for imbalanced data, like we have here. **From our business perspective, area under the PR Curve will be the most effective metric since this calculation puts more focus on the positive class rather than the negative class. When building our models (Section 4), we focused on maximizing the area under the PR Curve to choose the most optimal hyperparameters.**

1.4. Databricks Spark Cluster: Configuration

Parameter	Value	Description
Cluster Name	team12	Cluster name for the team
Custer Policy	Unrestricted	A cluster policy limits the ability to configure clusters based on a set of rules. Unrestricted policy allows to create fully-configurable clusters and does not limit any cluster attributes or attribute values.
Cluster Mode	High Concurrency	Databricks supports three cluster modes: Standard, High Concurrency, and Single Node. The default cluster mode is Standard. A High Concurrency cluster is a managed cloud resource. The key benefits of High Concurrency clusters are that they provide fine-grained sharing for maximum resource utilization and minimum query latencies.
Cluster Driver Size	Standard_D4s_v3 16 GB Memory, 4 Cores	The node type of the Spark driver
Cluster Worker Size	Standard_D4s_v3 16 GB Memory, 4 Cores	The node type of the Spark worker
Cluster Workers Quantity	min_workers: 1 max_workers: 6	Number of worker nodes that this cluster should have. Autoscaling is enabled.
Spark Version	9.1.x-cpu-ml-scala2.12	Version of Spark
Databricks Runtime Version	9.1 LTS ML (includes Apache Spark 3.1.2, Scala 2.12)	The runtime version of the cluster.

References

[1] Configure clusters

1.5. Background Research and Baselines

Prior research and professional papers have utilized advanced machine learning modeling techniques to also predict flight delays.

One study focused on predicting flight delays in Hong Kong, using five machine learning models: Random Forest, Logistics Regression, K-nearest Neighbors, Decision Tree, Naïve Bayes. In this study, random forests had the highest accuracy at 66.39% for predicting flight delays. [1]

In another study, Chakrabarty used a Gradient Boosting Classifier Model to predict whether American Airline flights in the top 5 airport stations in the United States would be delayed. Obtaining the best set of hyper-parameters using Grid Search, this model achieved a maximum accuracy of 85.73%. [2]

A study conducted in 2020 utilized Deep Learning methods and the Levenberg-Marquart algorithm to predict flight delays in the United States. Since the dataset used was imbalanced, this study used undersampling methods to create a balanced dataset. Their best model achieved an accuracy of 92.1% on the imbalanced dataset and 96.% on the balanced dataset. [3]

One non-research article included several reasons why flights were delayed. Describing different environmental factors and preparation issues, one important reason listed was called the "knock-on effect". This rotational delay is caused when the arriving aircraft arrived late, causing the next departure for the same aircraft to be delayed. We believe that this metric could be an important reason that causes flight delays, so we computed this metric in the feature engineereing section to be used in our model. [4]

References

[1] The Prediction of Flight Delay: Big Data-driven Machine Learning Approach

[2] A Data Mining Approach to Flight Arrival Delay Prediction for American Airlines

[3] Flight delay prediction based on deep learning and Levenberg-Marquart algorithm

[4] Why is My Flight Delayed? The 20 Main Reasons for Flight Delays

[5] Random Search in Spark ML

[6] Random Search for Hyper-parameter Optimization

1.6. Links to Notebooks

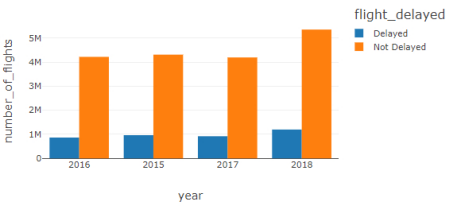
- Exploration
- Airlines Dataset: Transformation and Feature Engineering
- Weather Dataset: Transformation and Feature Engineering
- Joins
- Machine Learning Models
- Support Vector Machines: Native Implementation on RDDs
- 2020 Model Evaluation
  - 2020 Airlines Data: Transformation and Feature Engineering
  - 2020 Weather Data: Transformation and Feature Engineering
  - 2020 Model Evaluation: Joins, Pipeline Transformation and Model Evaluation

2. Exploratory Data Analysis

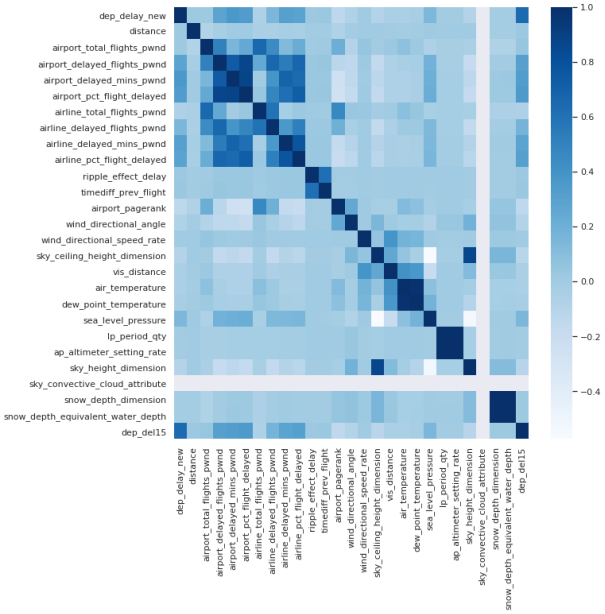
Final Project/Exploration: Full notebook that contains all the code and diagrams for our EDA

For the purposes of this report, we focused on key illustrations in our EDA process that helped drive our business motivations and inform ways to improve the problem-solving process. For a full plots and diagrams in our EDA, please refer to the notebook in the link above.

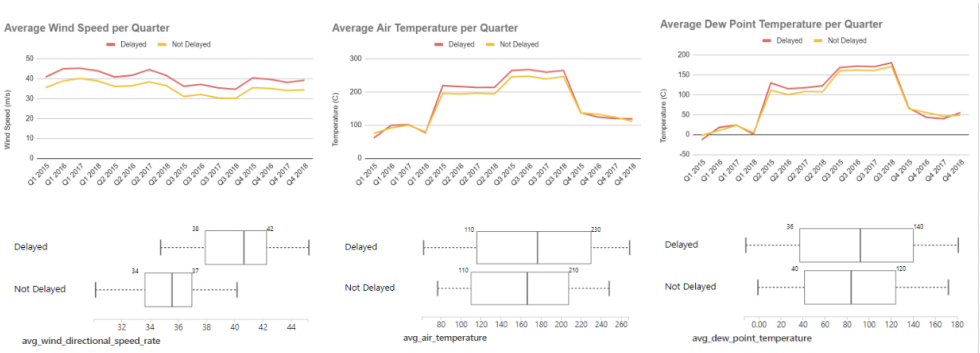
Generating the above cumulative density function (CDF) of the amount of time a flight was delayed gave an interesting insight into the delays. Approximately half of the delayed flights in the United States were delayed by 15 mins or more. This further confirms the value of our project to drive insights into what causes a flight to be delayed for more than 15 mins.



Per the documentation, flights delayed more than 15 minutes were classfied as delayed flights. We generated a visualization for the distribution of our target variable `DEP_DEL15` grouped by `YEAR`. As shown in the visualization above, the data set seemed to be quite imbalanced, illustrating that the number of non-delayed flights far outweighed the number of delayed flights each year. This was a crucial insight discovered, as this data imbalance can prove to be problematic in our model building process. In Section *4.1.1. Undersampling*, we outlined the undersampling technique to counteract this issue.



We also generated the correlation heatmap above to illustrate the idea if certain continuous features were correlated with each other. The correlation heatmap above also includes newly introduced features that were generated via feature engineering, as described in Section *3.1.2.*, *3.1.3.*, and *3.1.4.* Specifically, we were looking to see if certain features were correlated with our target variable `DEP_DEL15` or if features were correlated with each other. By discovering independent variables highly correlated with other independent variables, we could deploy a feature selection method to eliminate redudant variables, avoid multicollinearity, and improve model performance.



Initial data analysis show a quarterly average of weather metrics such as wind speed, air temperature and dew point temperature. We also created box plots in order to show interquartile range and median values for each weather-related metric for flights that occurred for 2015-2018. Intuitively, metrics for delayed flights should have faster wind speed, lower air temperatures and lower dew point temperatures (in case of freezing). We included these plots here as they show promising correlation / visual that weather-related data might help predict whether a flight will be delayed or not.

3. Preprocessing/Feature Engineering

Before we could build models and make predictions on the data, we first preprocessed the airlines and weather dataset, cleaning the data as much as possible. We also outlined feature engineered variables added in our dataset.

3.1. Airlines Dataset

Final Project/Airlines: Full notebook for all the code for preprocessing and feature engineering on the airlines dataset

For feature engineering, we split the 24 hour time window into 1 hour departure time blocks. We defined the prediction window time block as the 1 hour time block that is before the 2 hours of scheduled departure. We used these time blocks to calculate different performance metrics.

3.1.1. Cleaning the Data

For the purposes of this project, we decided to only focus on non-delayed and delayed flights. This meant that we filtered cancelled flights out of the data. Additionally, we removed rows with null values in our column of interest, `DEP_DEL`. There were a very small number of rows that had a null value in `ARR_TIME`, so we removed those columns. We also encountered duplicated rows, and we removed ~30M duplicate rows (identical flights) by matching values in `FL_TZ`, `ORIGIN_ICAO`, `DEST_ICAO`, and `TAIL_NUM`. There were 109 columns, many of which were filled with mostly null columns, columns not related to departure delay, or columns with information that was taken at or after departure delay. These such columns were not included when joining the datasets. We selected the following columns:

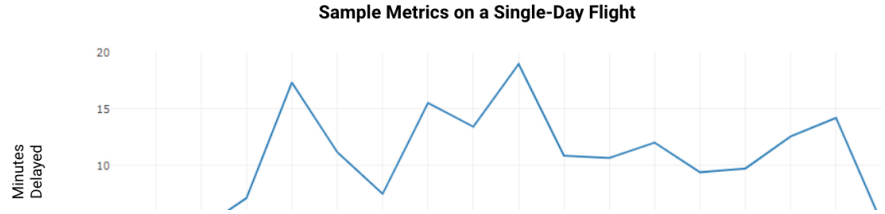
- `YEAR` : year in which the flight took place
- `QUARTER` : quarter of the year in which the flight took place
- `MONTH` : month in which the flight took place
- `DAY_OF_MONTH` : day of the month in which the flight took place
- `DAY_OF_WEEK` : day of the week in which the flight took place
- `FL_DATE` : date in which the flight took place
- `ORIGIN` : origin airport
- `ORIGIN_ICAO` : origin airport ICAO code
- `ORIGIN_CITY_NAME` : origin city name
- `ORIGIN_AIRPORT_ID` : origin airport ID
- `ORIGIN_STATE_ABR` : origin state abbreviation
- `DEST_AIRPORT_ID` : destination airport ID
- `DEST_STATE_ABR` : destination state abbreviation
- `DEST_CITY_NAME` : destination city name
- `DEST` : destination airport
- `DEST_ICAO` : destination airport ICAO code
- `OP_UNIQUE_CARRIER` : airline/carrier for the flight
- `OP_CARRIER_AIRLINE_ID` : airline/carrier ID for the flight
- `OP_CARRIER_FL_NUM` : flight number
- `TAIL_NUM` : aircraft number
- `DEP_TIME_BLK` : departure time block
- `ARR_TIME_BLK` : arrival time block
- `CRS_DEP_TIME` : scheduled departure time
- `CRS_ARR_TIME` : scheduled arrival time
- `CRS_ELAPSED_TIME` : estimated flight duration
- `DISTANCE` : distance of flight
- `DISTANCE_GROUP` : distance intervals, every 250 miles, for flight segment
- `DEP_DELAY_NEW` : difference in scheduled departure time and actual departure time, in minutes
- `DEP_DEL15` : binary indicator for non-delayed flight (0) or delayed flight (1)
- `DEP_TIME` : actual departure time
- `ARR_TIME` : actual arrival time
- `ARR_DEL_NEW` : difference in scheduled arrival time and actual arrival time, in minutes

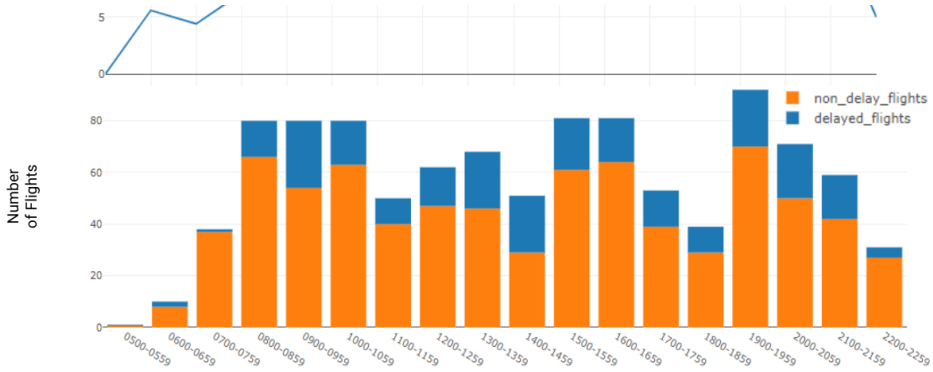
3.1.2. Feature Engineering: Performance of Airports and Airlines in Each Hour

We calculated metrics that illustrated the performance of the **origin airport** on the flight date and the departure window block. Combining these metrics with the airlines data using the prediction window time block, these new variables aimed to show how well the **origin airport** was performing at the time of prediction for each flight. The new features created were:

- `airport_total_flights_pwnd` : total flights at the origin airport at the time of the prediction window
- `airport_delayed_flights_pwnd` : number of flights delayed at the origin airport at the time of the prediction window
- `airport_delayed_mins_pwnd` : average delayed minutes of flights delayed at the origin airport at the time of the prediction window
- `airport_pct_flight_delayed` : percentage of flights that were delayed at the origin airport at the time of the prediction window

Here is an illustration of hourly metrics for Atlanta International Airport (ATL) on February 23, 2015. We used this hourly metric to associate knock-on effect for flights.



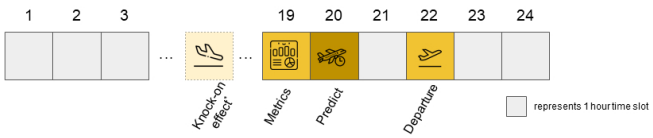


Similarly, we calculated metrics that illustrated the performance of the **airline** on the flight date and the departure window block. Combining these metrics with the airlines data using the prediction window time block, these new variables aim to show how well the **airline** was performing at the time of prediction for each flight. The new features created were:

- `airline_total_flights_pwnd` : total flights from the airline at the time of the prediction window
- `airline_delayed_flights_pwnd` : number of flights delayed from the airline at the time of the prediction window
- `airline_delayed_mins_pwnd` : average delayed minutes of flights delayed from the airline at the time of the prediction window
- `airline_pct_flight_delayed` : percentage of flights that were delayed from the airline at the time of the prediction window

3.1.3. Feature Engineering: Knock-on Effect (Late Arrival of Incoming Flight)

We investigated the Knock-On effect or delay due to late arrival of an incoming flight that could impact the departure of the next flight. This ripple effect can be understood by looking at the previous arriving flight with the same tail number (`TAIL_NUM`) that arrived at least 15 minutes late. In order to avoid data leakage, we required that there was at least a gap of 2 hours 15 minutes between the time of arrival and the next flight departure. The diagram below illustrates the timeframe of the knock-on effect:



The new features created were:

- `ripple_effect_delay` : departing aircraft's previous flight arrival delay in minutes
- `timediff_prev_flight` : time difference between scheduled departure time and scheduled arrival time of previous flight, in minutes

Knock-On Effect: Notebook with all the code and implementation of the Knock-on Effect

References

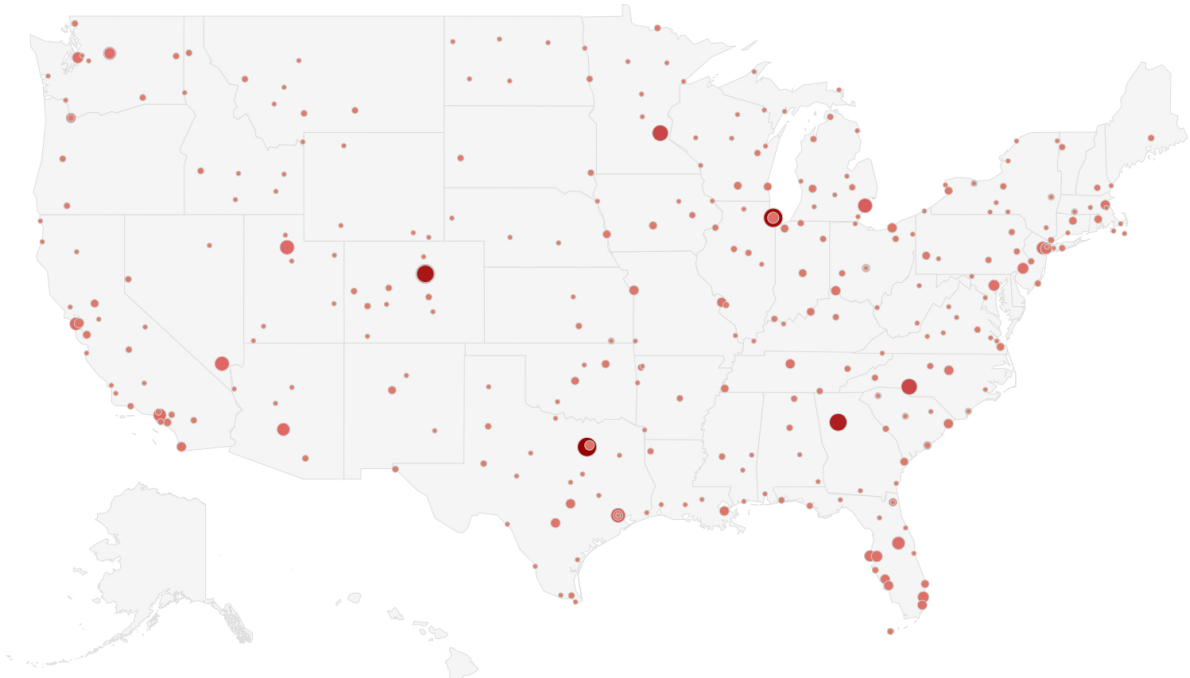
[1] Why is My Flight Delayed?

3.1.4. Feature Engineering: Pagerank of Flight Graph

We computed the pagerank for the airlines graph. First, we creates a `GraphFrame` object with each unique origin and/or destination airport as the vertex and a directed edge between two airports if there was a flight connecting them. `GraphFrame` library provides `pagerank` method that computes the pagerank of the unweighted graph. In the context of this project, pagerank serves as a *proxy* for airport movement. The new feature created was:

- `airport_pagerank` : pagerank of the origin airport

Here is the pagerank of each airport in the flight graph represented in the United States map, where the size of the dot is proportional to the pagerank of the corresponding airport. The size indicates that the flight movement in the airport and the importance of the airport in connectivity across the country.



Pagerank: Notebook with code and implementation of the pagerank

3.2. Weather Dataset

Final Project/Weather: Full notebook for all the code for preprocessing and feature engineering on the weather dataset

3.2.1. Selecting Relevant Data

Cleaning up and filtering the weather dataset was crucial to efficiently join the weather data with the airline data. We focused on:

- Filtering records on the country for the US only using the `NAME` column
- Filtering the records by `REPORT_TYPE` `FM-15` as that data was related to aviation.

After the cleanup, we selected only useful columns based on the EDA. The selected columns were `STATION`, `COUNTRY`, `DATE`, `LATITUDE`, `LONGITUDE`, `REPORT_TYPE`, `CALL_SIGN`, `WND`, `CIG`, `VIS`, `TMP`, `DEW`, `SLP`, `GA1`, `GE1`, `GD1`, `AA1`, `AJ1`, `AT1`, `IA1`, `MA1`, `fl_date`

3.2.2. Handling Nested Comma Separated Values

When conducting EDA on the weather data, we noticed many of the columns in the weather dataset contained multiple values in a comma separated format. To analyze these features properly and use them in our models we needed to extract these features from their comma separated format.

The features that were split are `WND`, `CIG`, `VIS`, `TMP`, `DEW`, `SLP`, `GA1`, `GE1`, `GD1`, `AA1`, `AJ1`, `AT1`, `IA1`, and `MA1`. The extracted features were based on the weather data documentation found here: Weather Documentation

Original Column	Extracted Features

`WND` - Wind Observation
NOAA - Page 7
	`wind_directional_angle`
	`wind_directional_qc`
	`wind_directional_type_code`
	`wind_directional_speed_rate`
	`wind_directional_speed_qc`

`CIG` - Sky Condition
NOAA - Page 9
	`sky_ceiling_height_dimension`
	`sky_ceiling_qc`
	`sky_ceiling_determination_code`
	`sky_ceiling_cavok_code`

`VIS` - Visibility Observation
NOAA - Page 10
	`vis_distance`
	`vis_distance_qc`
	`vis_variability_code`
	`vis_variability_qc`

`TMP` - Air Temperature NOAA - Page 10	`air_temperature` `air_temperature_qc`
`DEW` - Dew Point Temperature NOAA - Page 11	`dew_point_temperature` `dew_point_qc`
`SLP` - Sea Level Pressure NOAA - Page 12	`sea_level_pressure` `sea_level_pressure_qc`

`AA1` - Liquid precipitation
NOAA - Page 13
	`lp_period_qty`
	`lp_depth_dimension`
	`lp_condition_code`
	`lp_quality_code`
	
	* `lp` means liquid precipitation

`MA1` -Atmospheric pressure
NOAA - Page 88
	`ap_altimeter_setting_rate`
	`ap_altimeter_qc`
	`ap_station_pressure_rate`
	`ap_station_pressure_qc`
	
	* `ap` means atmospheric pressure

`GD1` - Sky Cover Summation State Identifiers
NOAA - Page 55
	`sky_coverage_code`
	`sky_coverage_code_2`
	`sky_coverage_qc`
	`sky_height_dimension`
	`sky_height_dimension_qc`
	`sky_characteristic_code`

`GE1` - SKY Condition Observation
NOAA - Page 9,57
	`sky_convective_cloud_attribute`
	`sky_vertical_datum_attribute`
	`sky_base_height_upper_range_attribute`
	`sky_base_height_lower_range_attribute`

`IA1` - Ground Surface Observation NOAA - Page 76	`ground_observation_code` `ground_observation_qc`

`AJ1` - Snow Depth Identifier
NOAA - Page 21
	`snow_depth_dimension`
	`snow_depth_condition_code`
	`snow_depth_qc`
	`snow_depth_equivalent_water_depth`
	`snow_depth_water_condition_code`
	`snow_depth_water_qc`

`AT1` - Daily Present Weather Observation
NOAA - Page 27
	`weather_obs_source_element`
	`weather_obs_weather_type_num`
	`weather_obs_weather_type_abb`
	`weather_obs_qc`

3.2.3. Imputing Missing Values

For un-signed columns, the values `99`, `999`, `9999`, `99999` or `999999` are used as the value where the data is missing. For the signed columns, the values `+99`, `+9999` or `+99999` are used as the value where the data is missing. The missing values are also indicated by condition code, discrepancy code or quality code `3`, `7` or `9`.

For the numeric features, the null and missing values were imputed using the median. We did not use mean as the distributions were skewed.

Imputation of the missing values was done **per fold** for cross validation and we used the imputed values for cross-fold validation. For the final model, we compute the imputed values across the training dataset (2015-2018) and use these values for training and on testing datasets.

3.3. Joins

3.3 Joins

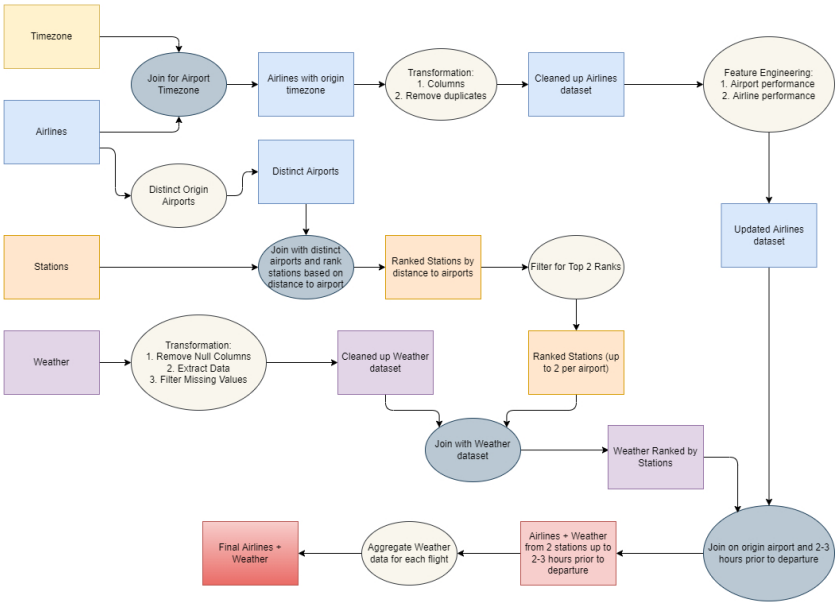
Final Project/Joins: Full notebook for all the code for joining the datasets together

We had to join the airlines data with the other datasets in order to collect all the features and execute the necessary pre-processing and feature engineering. We devised 4 main joins:

1. Airlines & Timezone: We joined the airlines with timezone so that we could calculate the local time at each of the airports. We used the timezone details to calculate the local departure time, local arrival time, etc.
2. Stations & Distinct airports: We joined the stations within 500 mile distance from the airports and then picked the top 2 closest stations to minimize the size of the data and picked only the stations that could help define the weather conditions around the airport.
3. Weather & Top 2 station per airport: We joined weather data with the top 2 stations to get all the weather features that are important for the model.
4. Airlines and Weather: This is the final join where we join the airlines data and the weather data. We picked only weather data acquired before 2 hours of departure. Aggregate the weather data for each flight as we have data from 2 stations.

Imputation of the missing values and pagerank calculation was done **per fold** for cross validation and we used the imputed values for cross-fold validation. For the final model, we compute the imputed values across the training dataset (2015-2018) and use these values for training and on testing datasets.

Below is a flowchart of our pipeline to join the data:



3.3.1. Repartition

We were using default partitioning parameters. Our first attempt to join the airlines and weather datasets ran for around 1.77 days. We optimized the joins by filtering the data using different steps as outlined in the Joins section to cut down on the amount of relevant data.

However, after looking at the documentation, we understood that by default we would have only 200 partitions, which could complicate the data processing. We then decided to pick `FL_DATE` as the partition key. We also set the `spark.sql.shuffle.partitions` to 1000 and the `spark.sql.files.minPartitionNum` to 1000 as we had 5 years of data. This helped improve the performance of our joins; the airlines and weather data join completed in only 2.79 minutes.

4. Classifiers and Novel Ideas

4.1. Cross Validation and Hyperparameter Tuning

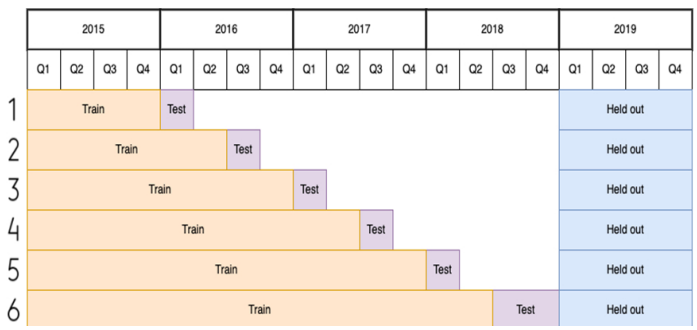
We utilized cross validation to evaluate the the performance of the model and avoid overfitting. In general, cross validation splits the dataset into several folds. In each fold, one portion is used to train the model and the other portion is used to test the model. However, with time-series data we need to take a slightly different approach as there is a temporal dependency between the data points that needs to be accounted for during the cross validation in order to avoid data leakage. We trained the model on a small set of data and then immediately tested the model on the set of data directly after the training data. Then in the next fold, we include the test data in the training data and then subsequent data sets immediately following were used for testing. We continue until all folds are complete.

Following this process, we trained and tuned our models on the data from 2015-2018 and then tested the best model against a heldout, never-before-seen dataset from 2019. We used this expanding window approach for cross-validation and custom cross validation[**1**]. In the first fold, we trained the model on all of the data in 2015 and test it on the immediate quarter after (2016 Q1). Then in the next fold, we included the test quarter in the previous fold (2016 Q1), added an additional quarter for training (2016 Q2), and used the subsequent quarter for testing (2016 Q3).

The result of this cross validation approach became a six-fold cross validation method. Please refer to the table and illustration below for a better understanding of our cross validation method:

Table 4.1

Fold	Training Dataset	Test Dataset
Fold 1	2015 Q1-Q4	2016 Q1
Fold 2	2015 Q1-Q4, 2016 Q1-Q2	2016 Q3
Fold 3	2015 Q1-Q4, 2016 Q1-Q4	2017 Q1
Fold 4	2015 Q1-Q4, 2016 Q1-Q4, 2017 Q1-Q2	2017 Q3
Fold 5	2015 Q1-Q4, 2016 Q1-Q4, 2017 Q1-Q4	2018 Q1
Fold 6	2015 Q1-Q4, 2016 Q1-Q4, 2017 Q1-Q4, 2018 Q1-Q2	2018 Q3-Q4

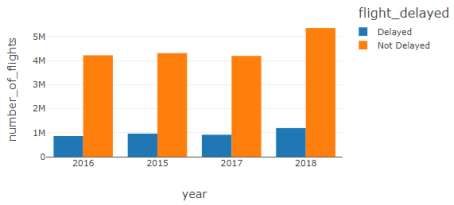


After the cross-validation , we pick the best hyperparameters based on the **Area under the PR Curve** metric as described in Section *1.3. Evaluation Metrics*.

Cross Validation: Notebook with the full code and implementation of the cross-validation.

4.1.1 Undersampling

In our EDA, we discovered that the number of non-delayed flights far outweighed the number of delayed flights, causing a data imbalance for our binary classification problem. This can be problematic as it would be much easier for models to predict for the majority class than the minority class. The greater the imbalance, the higher the bias of the model towards the majority class. In this problem, we feared that our models would struggle at predicting delayed flights. To rectify this issue, we undersampled non-delayed flights during the cross validation process in hopes of improving our model performance.



4.1.2. Grid Search and Random Search

Of the available strategies for hyper-parameter tuning using cross validation, we evaluated Grid Search and Random Search. In both cases, we provided a list of values for each hyperparameter. In Grid Search, cross-validation tested every possible combination of the listed values. So, time would still be spent evaluting on poor hyperparameter values. However, in Random Search, the process randomly searches the hyperparameter ranges based on the probabilities of given distribution. Here we are less likely to spend time on low performing hyper-parameter values. Thus, we observed that the cross validation time with Random Search was signficantly less than the cross validation time for Grid Search.

We used `RandomGridBuilder` to set the hyperparameters randomly from a distribution. The code is referenced from: [**2**]

References

[1] Creating a Custom Cross-Validation Function in PySpark

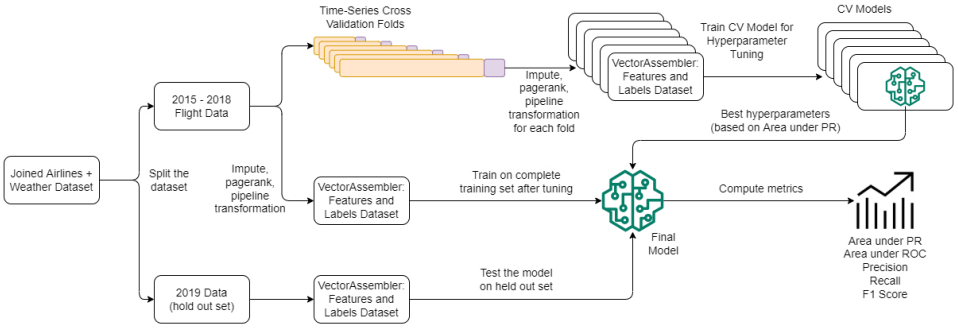
[2] Random Search

4.2. ML Pipeline Transformation

Our machine learning pipeline consists of the following steps.

- Split the data into training (2015-2018) and heldout (2019) datasets
- Perform cross validation as described in *Section 4.1 Cross Validation* for selecting the best hyperparameters based on the maximum **Area under the PR Curve**
- Train the model with the best hyperparemeters on the full training dataset (2015-2018)
- Evaluate the model on the 2019 heldout dataset

Please refer to the illustration below for a better understanding of our ML pipeline transformation:



During the cross validation process, we imputed the missing values and calculate airport pagerank for each fold before doing the pipeline transformation. The pipeline transformation includes the following steps.

1. `StringIndexer` for features that have string values.
2. `OneHotEncoder` for categorical features.
3. `VectorAssembler` to convert the features into a vectorized form that can be used in Spark ML models.
4. `StandardScaler` to normalize the features. Note that we set `withMean` to `False`. In other words, we do not scale the mean, only set the standard deviation to 1.

ML Pipeline transformation: Notebook with the code and implementation of the pipeline transformation.

Subsequently, we trained and evaluated the cross-validation model for each fold. Based on the highest **Area under the PR Curve** metric for the cross validation models, we selected the hyperparameters from the best model.

Once we picked the hyperparameters, we imputed the missing values, calculated airport pagerank, and executed the pipeline transformation on the full training dataset. Then, we trained the model on the full training dataset and evaluated the results on the 2019 heldout dataset.

4.3. Baseline Model: Logistic Regression with CV Based Hyperparameter Tuning

Logistic regression is a popular choice for categorical predictions. In this problem, we will be utilizing binomial logistic regression, since our outcome variable `DEP_DEL15` is binary.

4.3.1. Model Theory

As with many machine learning methods, logistic regression can be formulated as a convex optimization problem, which is the task of finding a minimizer of a convex function f that depends on a variable vector w or weights, having d entries. The objective function is of the form:

$$f(w) := \lambda R(w) + \frac{1}{n}\Sigma_{i=1}^{n} L(w; x_i, y_i)$$

where $x_i \in R^d$ are training data for $1 \leq i \leq n$ and $y_i \in R$ are the labels. For logistic regression, the loss function $L(w; x_i, y_i)$ can be expressed as: $L(wv; x, y) := \log(1 + \exp(-ywv^T x))$.

Given a new data point x, the logistic regression model makes predictions using the logistic function:

$$f(z) = \frac{1}{1 + e^{-z}}, \text{where } z = wv^T x$$

The output of this equation $f(z)$ can be interpreted as the probability that that the new data point x has a positive label.

Reference: Spark Documentation (Classification), Spark Documentation (Linear Methods)

4.3.2. Hyperparameters

We used cross validation to tune the following hyperparameters for the model.

Hyperparameter	Description	Search Values
fitIntercept	should the model fit an intercept term	$[True, False]$
regParam	regularization parameter	[0, 1]
maxIter	maximum number of iterations	[0, 100]

```
lr_estimator = LogisticRegression(featuresCol='features', labelCol='label')

lr_random_grid = RandomGridBuilder(5) \
                 .addDistr(lr_estimator.fitIntercept, lambda: bool(np.random.choice([False, True]))) \
                 .addDistr(lr_estimator.regParam, lambda: np.random.rand()) \
                 .addDistr(lr_estimator.maxIter, lambda: np.random.randint(100)) \
                 .build()

lr_evaluator = BinaryClassificationEvaluator(metricName='areaUnderPR')
```

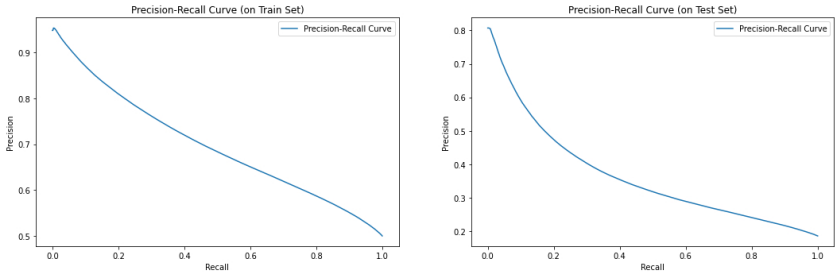
Running a random search (with 5 models per fold) for hyperparameters provided us with the best hyperparameter: `{'fitIntercept': False, 'regParam': 0.252181702918517, 'maxIter': 52}`. Cross validation took 2.44 hours to complete.

4.3.3. Training and Evaluation

Results on 2019 Heldout Dataset

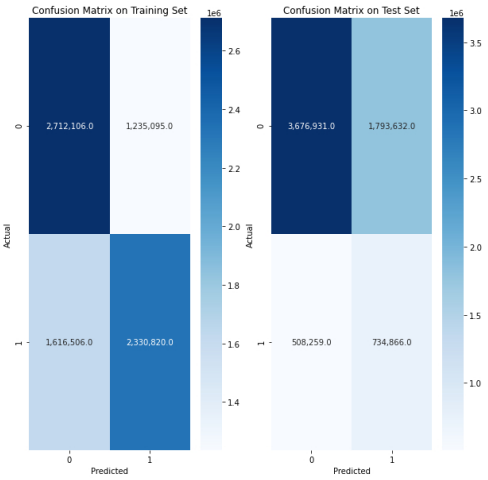
Metric	Training	Test
Area Under ROC	0.694060	0.685481
Area Under PR	0.697966	0.362964
F1 score	0.620457	0.389682
Recall	0.590481	0.591144
Precision	0.653639	0.290633
Accuracy	0.638788	0.657135

Area Under PR



For other metrics, please refer to the notebook.

Confusion Matrix



4.4. Decision Trees

Decision trees are a non-parametric supervised learning method. In very simple terms, it makes its decisions using tree-like models, where the beginning node splits into series of feature-based splits that end with decisions and predictions as leaves. The model outputs are relatively easier to interpret, can handle categorical features, extend to multiclass classification setting, do not require feature scaline, and able to capture non-linearities and feature interactions.

4.4.1. Model Theory

The decision tree is a greedy algorithm that performs a recursive binary partitioning of the feature space. Each decision is made greedily by selecting the best split from a set of possible splits, in order to maximize the information gain at a tree node.

The node impurity is a measure of homogeneity of the labels at the node. By default, the impurity hyperparameter used for our classification model is Gini Impurity.

$$\sum_{i=1}^{C} f_i(1 - f_i)$$

$$\sum_{i=1} f_i(1 - f_i)$$

where f_i is the frequency of label i at a node and C is the number of unique labels

The information gain is the difference between the parent node impurity and the weighted sum of the two child node impurities. Assuming that a split s partitions, the dataset D of size N into two datasets -- D_{left} with size N_{left}, and D_{right} with size N_{right} -- the information gain (IG) is as follows:

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

Reference: Decision Trees, Spark Documentation

4.4.2. Hyperparameters

For the hyperparameters, we have only changed `maxDepth` and `maxBins`

We used cross validation to tune the following hyperparameters for the model.

Hyperparameter	Description	Search Values
maxDepth	maximum depth of the tree	[2, 24]
maxBins	Max number of bins for discretizing continuous features	[2, 100]

```
tree_estimator = DecisionTreeClassifier(featuresCol='features', labelCol='label')

tree_param_grid = ParamGridBuilder() \
    .addGrid(tree_estimator.maxDepth, [2, 5, 10]) \
    .addGrid(tree_estimator.maxBins, [10, 20, 40]) \
    .build()

tree_random_grid = RandomGridBuilder(5) \
               .addDistr(tree_estimator.maxDepth, lambda: np.random.randint(2, 24)) \
               .addDistr(tree_estimator.maxBins, lambda: np.random.randint(2, 100)) \
               .build()

tree_evaluator = BinaryClassificationEvaluator(metricName='areaUnderPR')
```

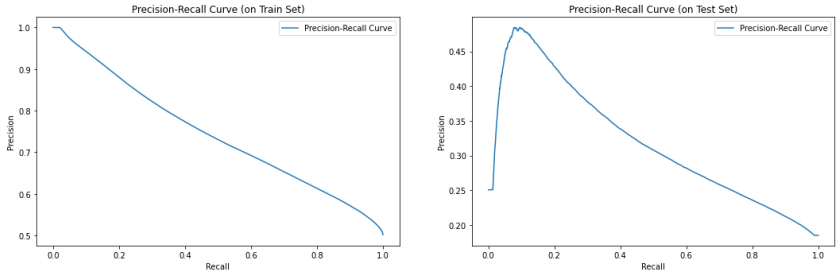
Running a random search (with 5 models per fold) for hyperparameters provided us with the best hyperparameter: `{'maxDepth': 18, 'maxBins': 12}`. Cross validation took 3.32 hours to complete.

4.4.3. Training and Evaluation

Results on 2019 Heldout Dataset

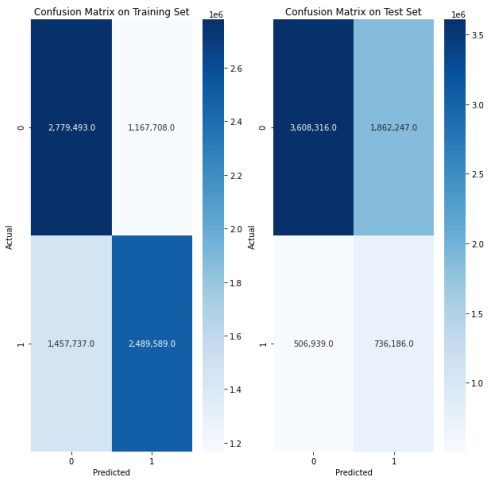
Metric	Training	Test
Area Under ROC	0.733856	0.673298
Area Under PR	0.744028	0.317968
F1 score	0.654757	0.383275
Recall	0.630703	0.592206
Precision	0.680718	0.283319
Accuracy	0.667435	0.647111

Area Under PR



For other metrics, please refer to the notebook.

Confusion Matrix



4.5. Gradient Boosted Trees

Gradient Boosted Trees are ensembles of iteratively trained decision trees, and is used in both regression and classification problems. Like other boosting methods, gradient boosting combines weak 'learners' into a single learner in iteration in order to minimize the loss function.

4.5.1. Model Theory

Gradient Boosting are ensembles of decision trees as they iteratively trains decision trees sequentially. At each iteration, the model uses the current ensemble of trees to predict the label of each training instance and these predictions are evaluated against the true labels. The loss function is used to put more emphasis on poor predictions so that in the next iterations, the model can account for these mistakes and reduce the loss function on the training data.

In general, objective function should always be defined with both training loss and regularization term:

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$

where L is the training loss function and Ω is the regularization term. The training loss measuers how predictive the model is with respect to training data. The regularization term controls the complexity of the model, which helps avoid overfitting.

For this binary classification problem, we used the Log Loss function, which is twice binomial negative log-likelihood.

$$2\sum_{i=1}^{n} log(1 + e^{-2y_i F(x_i)})$$

where N is the number of instances i, y_i is the label of a particular instance, x_i is the number of features in instance and $F(x_i)$ is the model's predicted label for the instance.

More on the theory on Gradient Boosted Trees explained in XGBoost section below.

Reference: Gradient Boosted Trees, Spark Documentation.

4.5.2. Hyperparameters

We used cross validation to tune the following hyperparameters for the model.

Hyperparameter	Description	Search Values
maxDepth	maximum depth of the tree	[2, 10]

```
gbtree_estimator = GBTClassifier(
     featuresCol='features',
     labelCol='label',
     maxIter=5,
     maxBins=32
)

gbtree_random_grid = RandomGridBuilder(2)
    .addDistr(gbtree_estimator.maxDepth, lambda: np.random.randint(10)) \
    .build()

gbtree_evaluator = BinaryClassificationEvaluator(metricName='areaUnderPR')
```

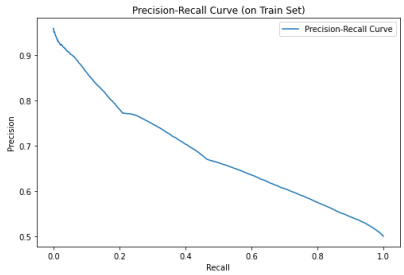
Running a random search (with 2 models per fold) for hyperparameters provided us with the best hyperparameter: `{maxDepth = 2}`. Cross validation took 11.95 minutes to complete. Note that he number of hyperparameters and the number of models we built for this classifier is much lesser than logistic regression model.

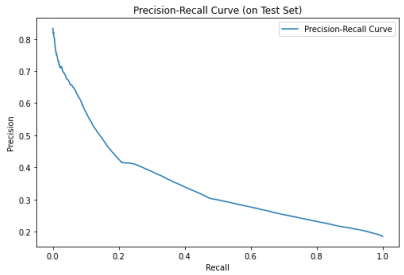
4.5.3. Training and Evaluation

Results on 2019 Heldout Dataset

Metric	Training	Test
Area Under ROC	0.678842	0.670607
Area Under PR	0.684207	0.346511
F1 score	0.573082	0.375564
Recall	0.505789	0.515503
Precision	0.661029	0.295380
Accuracy	0.623206	0.682592

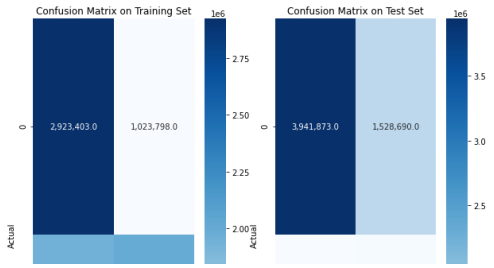
Area Under PR

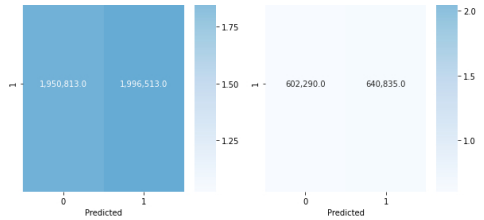




For other metrics, please refer to the notebook.

Confusion Matrix





4.6. XGBoost

XGBoost, which stands for "Extreme Gradient Boosting", is an optimized Gradient Boosting Machine Learning library. It is very similar to the gradient boosted trees algorithm, but the main differences are that XGBoost utilizes penalties like L1 and L2 regularization and trains quicker than a traditional gradient boosted algorithm because it takes advantage of parallelization across clusters. For modeling and naming purposes, think of XGBoost as a more "regularized" gradient boosting. It is a *much better* implementation of Gradient Boosted Trees to better accommodate raidly increasing size of datasets. This is especially useful for distributed training for large scale ML operations.

4.6.1. Model Theory

XGBoost follows a very similar algorithm to gradient boosted trees. Similarly, it is also used for supervised learning problems where training data with multiple features, x_i, is used to predict a target variable, y_i, which in our case is whether a flight will be delayed or not.

The model choice of XGBoost is decision tree ensembles consisting of a set of classification and regression trees (CART). Prediction scores for each individual tree are summed up to get the final score. Mathematically, this can be written in the form:

$$\hat{y}_i = \sum_{i=1}^{K} f_k(x_i),\ f_k \in F$$

where K is the number of trees, f_k is a function in the functional space F which is the set of all possible CARTs. From the original paper, the objective function is defined, before reformulation, to be:

$$\text{obj}(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \omega(f_k)$$

where $\omega(f_k)$ is the complexity of the tree f_k. In XGBoost, the complexity is defined as:

$$\omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

After reformulation objective function we want to optimize at each "step" of the t-th tree is

$$\text{obj}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

where $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$

References

[1] Introduction to Boosted Trees, XGBoost Documentation

4.6.2. Hyperparameters

We used cross validation to tune the following hyperparameters for the model.

Hyperparameter	Description	Search Values
max_depth	maximum depth of the tree	[2, 10]
n_estimators	number of estimators	[10, 200]

```
xgboost_estimator = XgboostClassifier(featuresCol='features',
                                      labelCol='label',
                                      missing=0.0,
                                      eval_metric='logloss')

xgboost_random_grid = RandomGridBuilder(3) \
                      .addDistr(xgboost_estimator.max_depth, lambda: np.random.randint(2, 10)) \
                      .addDistr(xgboost_estimator.n_estimators, lambda: np.random.randint(10, 200)) \
                      .build()

xgboost_evaluator = BinaryClassificationEvaluator(metricName='areaUnderPR')
```

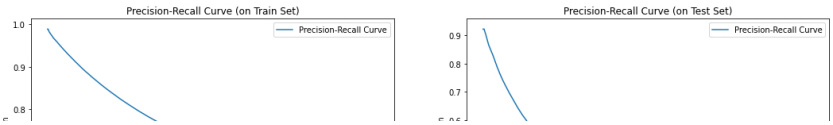
Running a random search (with 3 models per fold) for hyperparameters provided us with the best hyperparameter: `{'max_depth': 6, 'n_estimators': 32}`. Cross validation took 1.12 hours to complete. Note that he number of hyperparameters and the number of models we built for this classifier is much lesser than logistic regression model.

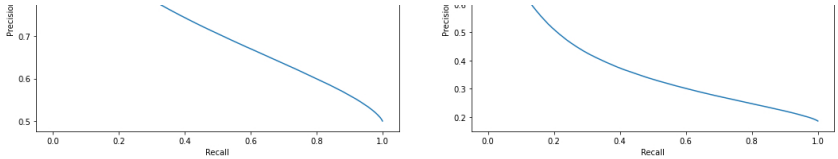
4.6.3. Training and Evaluation

Results on 2019 Heldout Dataset

Metric	Training	Test
Area Under ROC	0.712501	0.699037
Area Under PR	0.717967	0.386705
F1 score	0.638829	0.400001
Recall	0.614620	0.612218
Precision	0.665024	0.297037
Accuracy	0.652511	0.659922

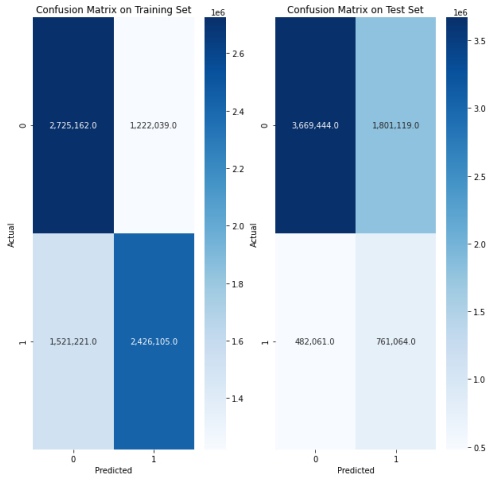
Area Undder PR





For other metrics, please refer to the notebook.

Confusion Matrix



4.7. Novel Ideas

In this project, we implemented the following novel ideas: (1) pagerank of the flight graph, (2) identification of knock-on effect and its impact on flight delays, and (3) evaluation of the trained models on 2020 dataset. Two of the ideas, namely, pagerank and knock-on effect, are focused on feature engineering and deriving new features from the data. While the third idea is to understand how effective is a model that is trained on 2015-2018 dataset against 2020 dataset when there are potentially other predictors that could contribute to the flight delays.

4.7.1. Pagerank

As discussed in Section 3.1.4, we implemented unweighted pagerank computation of the flight graph using the 2015-2018 dataset. We used `GraphFrame` to construct the flight graph and `pagerank` method to compute the pagerank of each vertex (i.e., each airport) in the graph. This pagerank of origin airport is then augmented to the flight dataset. Pagerank of the origin airport serves as a signal for the flight movement in that airport.

4.7.2. Knock-on Effect

As discussed in Section 3.1.3, we implemented knock-on effect determination for a flight. If the incoming flight is delayed then it is possible that the departing flight following that will be delayed. To avoid leakage of data, we consider knock-on effect only for flights whose incoming flight (with the same aircraft, as determined by the `tail_num`) is at least 2 hours and 15 minutes. Knock-on effect serves as a signal for the delay in the departure of the outgoing flight.

4.7.3. Model Evaluation on 2020 Dataset

We evaluated our models on 2020 dataset as well. The model was trained on 2015-2018 data and evaluated on 2020 dataset. Here are our steps in this process:

1. Data collection
2. Data cleaning and feature engineering
3. Joins
4. Pipeline transformation
5. Model evaluation and metrics

Data Collection

We collected the 2020 flight dataset from BTS.gov website for each month of the year. The data was available in CSV format. We uploaded the data to our blob storage. Additionally, we collected the 2020 weather dataset from NOAA website for all the stations that NOAA tracks. Again, we uploaded this data to our blob storage.

Note that we did not perform any EDA on this dataset as we wanted to evaluate our existing models on this dataset. Our assumption is that how would our model fare against external factors such as delays due to COVID-19 restrictions.

Data Cleaning and Feature Engineering

The data cleaning and feature engineering steps are similar to the steps identified in Sections 3.1 and 3.2.

For the airlines dataset, we executed the following steps:

1. Join with timezone based on the origin airport of each flight
2. Filtering: remove duplicates, cancelled flights and flights will null `ARR_TIME`
3. Feature Engineering:
    1. Convert timestamp to UTC timestamp based on timezone information
    2. Identify the 1-hour block for departure time
    3. Derive airport and airline performance metrics for each 1 hour block
    4. Augment the flight data with the airport and airline performance metrics in the 1 hour block prior to the time we need to predict the outcome

For the weather dataset, we executed the following steps:

1. Filter for US stations and FM-15 report type.
2. Find distinct origin airports from the flight dataset and identify top 2 stations that are within 500 miles to the origin airport.
3. Join the stations with weather data and aggregate for each airport.

Joins

As mentioned in Section 3.3, we perform the join operation on the 2020 dataset. Here are the high-level steps:

1. Join flight data and weather dataset. Each flight data will have at most 2 weather stations
2. Aggregate weather data for each flight by computing average, min and max values for weather features.

Pipeline Transformation

Finally, we transform the dataset that can be used in our models. Specifically,

1. Fill missing values using the imputed values computed from the training dataset.
2. Add pagerank column for each flight based on the origin airport. The pagerank is compute on the training dataset.
3. Transform the features into vectorized format using the pipeline transformation. Again, this is the same step as identified in Section 4.2

Model Evaluation and Metrics

The model parameters for each of our models are checkpointed in our blob storage. We loaded these models and evaluated them again the 2020 dataset. Here are the results:

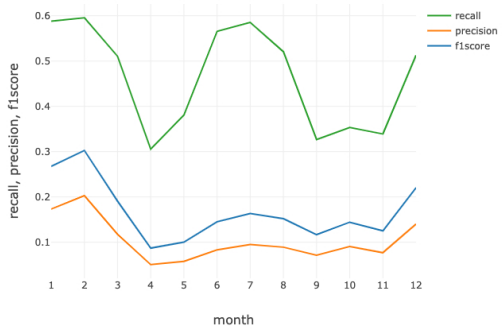
Metric	Logistic Regression	Descision Trees	Gradient Boosted Trees	XGBoost
Area Under ROC	0.588448	0.608837	0.618443	0.644644
Area Under PR	0.108296	0.136206	0.170174	0.195725
F1 score	0.197873	0.204086	0.214192	0.226878
Recall	0.514534	0.373189	0.265589	0.386395
Precision	0.122490	0.140446	0.179462	0.160583
Accuracy	0.625559	0.738726	0.825081	0.763625

As expected, these results show that our models perform poorly on 2020 dataset. Clearly, 2020 is an anomalous year with respect to the data under consideration. There are other forces that might have contributed to the airline on-time performance in 2020 compared to other years. Our precision is around 12-18% across our models with area under PR curve around 10-20%. Whereas, in 2019 dataset, our precision was around 30% and area under PR curve was around 32-38%.

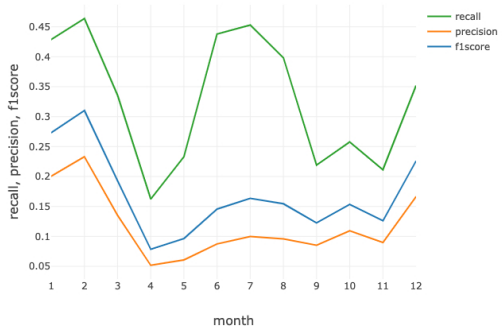
How did the model perform each month in the year 2020?

2020 posed an interesting challenge with respect to prediction of flight delays. For the first 3 months of 2020, we expect the prediction to be similar to earlier years. However, the COVID-19 pandemic triggered a global lockdown starting in the month of March. As a result, the number of flights started to go down and the on-time performance of those flights also got impacted. As we can see from the plots below, our model metrics suffered, especially, since the start of the pandemic. This suggests that the features used by our models are not sufficient for the prediction.

Logistic Regression

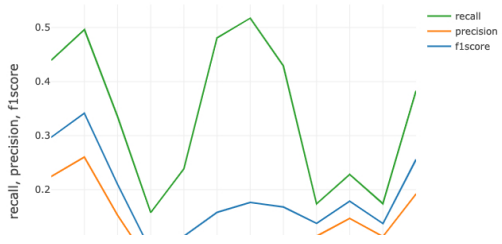


Decision Trees



Gradient Boosted Trees

XGBoost



4.8. Scalability Challenges and Solutions

During this assignment, we encountered scability challenges and creatively solved them to consume less time and speed up our computations:

- **Joining the Datasets**: Our initial attempt took about 1.77 days to join, which was highly inefficient. We solved this issue by adjusting the partitions as described in Section 3.3.1. The the final join took only 2.79 minutes.
- **Efficient Use of Memory**: During our computations, we checkpointed dataframes and wrote them to storage so that we would not have to rerun any computationally-heavy processes. We also cached/persisted as much as possible and unpersisted unused dataframes.
- **Grid Search vs Random Search**: During our cross validation process, we employed Grid Search and Random Search to tune our hyperparameters for our models. We first used Grid Search, but this process took more than 5 hours to find the best hyperparameters. To speed up the hyperparameter-tuning process, we used Random Search, which cut down the time by several hours. Greater detail of the differences between Grid Search and random Search is discussed in Section *4.1.2.*

5. Algorithm Theory: Support Vector Machines (SVM)

Support Vector Machine (SVM) is a supervised machine learning model that deals with classifying, regressing and detecting outliers. For this example, we are using SVM on a toy dataset to classify whether a flight will be delayed or not. In essence, SVM creates a *separation* by finding a hyperplane that separates two classes in a dataset.

Per the Spark documentation, Linear SVM is a standard method for large-scale classification tasks. As with many machine learning methods, Linear SVMs can be formulated as a convex optimization problem, which is the task of finding a minimizer of a convex function f that depends on a variable vector w or weights, having d entries. The objective function is of the form:

$$f(w) := \lambda R(w) + \frac{1}{n}\sum_{i=1}^{n} L(w; x_i, y_i)$$

where $x_i \in R^d$ are training data for $1 \leq i \leq n$ and $y_i \in R$ are the labels. For Linear SVMs, $L(w; x_i, y_i)$ can be expressed as a function of $w^T x$ and y.

$$\frac{\lambda}{2}||w||^2 + \frac{1}{n}\sum_{i=1}^{n}(1 - y_i(w^T x_i + b))$$

The objective function has two parts -- the regularizer that controls the complexity of the model, and the loss that measures the error of the model on the training data. The loss function $L(W; .)$ is typically a convex function in w. The fixed regularization parameter $\lambda \geq 0$, defines the tradeoff between the two parts of minimizing the loss (training error) and minimizing model complexity (avoiding overfitting). More specifically the loss function for Linear SVM is a hinge loss:

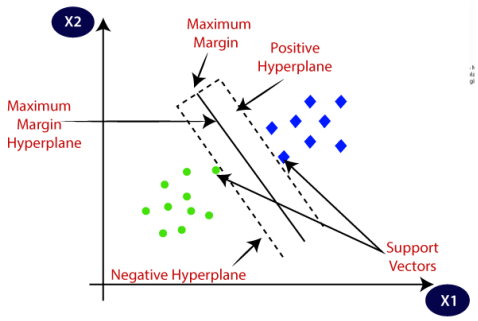
$$L(w; x, y) := \max(0, 1 - yw^T x), y \in (-1, +1)$$

Once we have the hyperplane, we can use the hyperplane to make predictions, h such that:

$$h = 1, \text{ if } w^T x \geq 0$$

$$h = -1, \text{ if } w^T x < 0$$

Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class, called functional margin. The larger the margin, the lower the generalization error of the classifier. The goal in SVM is to maximize the distance between the hyperplane and the data points, or the ' margin' as outlined in the figure below. (Source of the image: https://editor.analyticsvidhya.com/uploads/567891.png)



Gradient Descent

The optimization problem can we written as follows:

$$\min_{w} J(w), \text{where } J(w) = \frac{\lambda}{2}||w||^2 + \frac{1}{n}\sum_{i=1}^{n}(1 - y_i(w^T x_i + b)$$

This quadratic optimization problem can solved using gradient descent approach. Here is the framework for gradient descent:

- Initialize w
- Repeat the following steps until convergence or upto a maximum number of iterations:
  - Compute the gradient of the loss function of J at w. The gradient of loss function is:

$$\nabla J(w) = \lambda w, \text{if } y_i(w^T x_i + b) > 1$$

$$\nabla J(w) = \lambda w - y_i x_i, \text{otherwise}$$

  We compute the partial gradients for each training sample x_i and take an average across all samples.
  - Update the weights: $w = w - \frac{1}{n}\eta(\nabla J(w))$, where η is the learning rate of the gradient descent. Note that that the objective function includes L2 regularization term with (\lambda) as the regularization parameter.

Prediction

Once the weights are determined, we can use the model to predict samples. Note that for binomial classification, the labels are -1 and $+1$. To classify a sample, say x_j, we compute $w^T x_j + b$. If $(w^T x_j + b) >= 0$ then the sample is classified with label $+1$. Otherwise, the sample is classified with label -1.

References

[1] Stanford: CS 229 Lecture by Andrew Ng

[2] MIT: An Idiot's Guide to Support Vector Machines by R. Berwick

[3] Support Vector Machines: Training with Stochastic Gradient Descent by Vivek Srikumar

5.1 Illustration with a Toy Example

In this section, we will outline the math for one iteration of gradient descent algorithm to compute the model weights. As defined in SVM Example, consider the following dataset. The dataset consists of 2 features x_1, x_2.

1. Positive labeled samples (i.e., $y = 1$): $(3, 1, 1), (3, -1, 1), (6, 1, 1), (6, -1, 1)$
2. Negative labeld samples (i.e., $y = -1$): $(1, 0, 1), (0, 1, 1), (0, -1, 1), (-1, 0, 1)$

> Note that each training example is augmented with 1 to make the math simple (for dealing with bias terms)

Let's assume regularization parameter, $\lambda = 1$ and learning rate, $\eta = 1$.

Let us define the initial weights as follows: $w = (0, 0, b = 0)$, where the last term is the bias term. First, we compute the gradient of the loss function at this w. For the postive labeled training example $(3, 1, 1)$, $y_i(w^T x_i) = 0$. As a result, partial radient at this training example is: $(\lambda w - y_i x_i) = (0, 0, 0) - (3, 1, 1) = (-3, -1, -1)$. Similarly, for negative labeled training example $(1, 0, 1)$, $y_i(w^T x_i) = (-1)(0) = 0$. As a result, partial gradient at this example: $(\lambda w - y_i x_i) = (0, 0, 0) - (-1)(1, 0, 1) = (1, 0, 1)$.

The following table illustrates the first iteration where we look at each example and compute the subgradient.

Training Example Label	Training example data	Partial gradient
1	$(3, 1, 1)$	$(0, 0, 0) - (3, 1, 1) = (-3, -1, -1)$
1	$(3, -1, 1)$	$(0, 0, 0) - (3, -1, 1) = (-3, 1, -1)$
1	$(6, 1, 1)$	$(0, 0, 0) - (6, 1, 1) = (-6, -1, -1)$
1	$(6, -1, 1)$	$(0, 0, 0) - (6, -1, 1) = (-6, 1, -1)$
-1	$(1, 0, 1)$	$0, 0, 0) - (-1)(1, 0, 1) = (1, 0, 1)$
-1	$(0, 1, 1)$	$0, 0, 0) - (-1)(0, 1, 1) = (0, 1, 1)$
-1	$(0, -1, 1)$	$0, 0, 0) - (-1)(0, -1, 1) = (0, -1, 1)$
-1	$(-1, 0, 1)$	$0, 0, 0) - (-1)(-1, 0, 1) = (-1, 0, 1)$

Finally, we compute the update for the weight by taking an average of all the partial gradients, which is equal to $(-2.25, 0, 0)$. Since the learning rate $\eta = 1$, we update $w = (0, 0, 0) - (-2.25, 0, 0) = (2.25, 0, 0)$.

We repeat the process until we converge on the weights or a maximum number of iterations are reached.

6. Implementation

We implemented SVM for linear kernels natively using RDDs. In this section, we outline our implementation, define a small dataset for training this model, and evaluate the performance of the model on a test dataset. In addition, we also implemented cross validation to tune the learning rate of our algorithm. The hyperparameter is selected based on the model that provides the best area under the PR curve. Finally, we compare our implementation with Spark MLLib implementation (`SVMWithSGD`).

SVM Implementation: Notebook with all the code and the implementation of the gradient descent for SVM.

6.1. Gradient Descent

In our implementation, we first augmented each training example with 1 to make the math simple for dealing with bias terms in the weights. Then, we computed the partial gradients for each training example. Note that this computation occurs in parallel across all training samples. Once we have the partial gradients, we computed the mean of these partial gradients to determine the update for the weights. In addition, we also computed the regularization term and added it to the update. The weight is finally updated as $w = w + \eta(\lambda w + \text{mean of all partial gradients})$, where λ is the regularization parameter and η is the learning rate.

```
def GDUpdate(dataRDD, W, learningRate = 0.01, regParam = 0.1):
    """
    Perform one OLS gradient descent step/update.
    Args:
        dataRDD - records are tuples of (y, features_array)
        W       - (array) model coefficients
    Returns:
        update - update to the model
    """

    # add bias term
    augmentedData = dataRDD.map(lambda x: (x[1], np.append([1.0], x[2])))

    # broadcast the current weights / model
    W_broadcast = sc.broadcast(W)

    # spark job to compute the gradient

    def gradient(row):
        y, X = row[0], row[1]
        grad = 0.0
        # add gradient only for misclassified
        if y*np.dot(X, W_broadcast.value) < 1:
            grad = -1 * y * X
        return grad

    g = augmentedData.map(gradient).mean()

    Wreg = W * 1
    # do not regularize bias term
    Wreg[0] = 0

    update = learningRate * (g + regParam * Wreg)

    return update
```

6.2. Loss Function

Optionally, we also computed the Hinge loss at each iteration of the gradient descent algorithm. Like before, we augmented each training example with 1 to make the math simple for dealing with bias terms in the weight. For each training example, we computed a dot product of the weights and the training example. If the result had the same sign as the label, then the loss for that example is 0. Otherwise it is 1. This calculation is done in parallel across all training examples. And, finally, a mean loss is calculated using the mean function.

```
def hingeLoss(dataRDD, W):
    """
    Perform one OLS gradient descent step/update.
    Args:
        dataRDD - records are tuples of (y, features_array)
        W       - (array) model coefficients
    Returns:
        loss - hingeLoss of the current model for the data
    """

    # add bias term
    augmentedData = dataRDD.map(lambda x: (x[1], np.append([1.0], x[2])))

    # broadcast the current weights / model
    W_broadcast = sc.broadcast(W)
```

```
    def loss(row):
        y, X = row[0], row[1]
        z = y * np.dot(W_broadcast.value, X)
        hl = np.maximum(0, 1-z)
        return hl

    hinge = augmentedData.map(loss).mean()

    return hinge
```

6.3. Dataset

To test our implementation, we took the first quarter of 2015 data from the fully joined airlines and weather dataset. We split the dataset into a training set and a test set. The first 2 month of the dataset forms the training set, while the last month is the test set.

Features

In this dataset, we only considered the following features.

Date features: including month, day of week, day of month, and departure window (in units of 1-hour intervals).

Airline features: including distance, airport performance metrics in the 1-hour interval prior to prediction, airline performance metrics at the origin airport in the 1-hour interval prior to prediction, knock-on effect, and pagerank.

Weather features: including the average measures from the top 2 stations closest to the airport for wind directional angle, wind directional speed rate, sky ceiling height, temperature, dew point temperature, etc.

Transformation

We imputed the missing values using the training dataset (first 2 months). In addition, we computed pagerank of the flight graph using the training dataset. Subsequently, we performed pipeline transformation similar to the steps identified in Section 4.2.

RDD

Once we generated the vectorized features, we retrieve the RDD and use the RDD to train our model.

6.4. Cross Validation

We implemented cross validation to tune learning rate hyperparameter. Towards this end, we use training data from January 01, 2015 to February 15, 2015 as the training set and the data from February 16, 2015 to February 28, 2015 as the validation set. We computed the missing values by imputing the data using the training set in the cross validation fold. Similarly, pagerank is also computed only with the training set in the cross validation fold. Missing values and pagerank in the validation set are updated based on the training set in the cross validation fold.

We use area under the PR curve as the metric to pick the best hyperparameter.

6.4.1. Hyperparameters

We used cross validation to tune the following hyperparameters for the model.

Hyperparameter	Description	Search Values
learning_rate	learning rate for gradient descent	[0.01, 0.001, 0.0001]

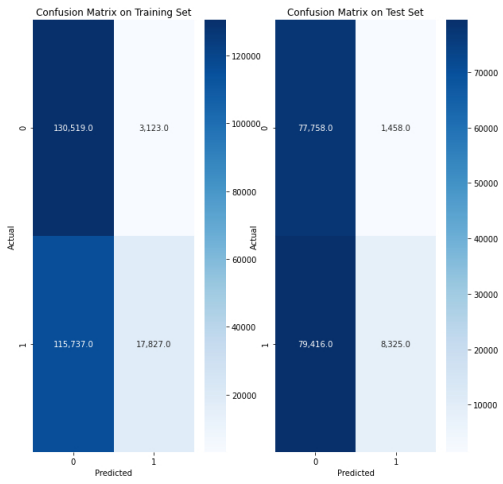
The best learning rate from cross validation is 0.01. We train a model using the 2 month dataset (January 1, 2015 to February 28, 2015) and test on the last month dataset (March 2015).

6.5. Model Training and Evaluation

Metrics

Metric	Training	Test
Area Under ROC	0.555052	0.538238
Area Under PR	0.698822	0.703687
F1 score	0.230749	0.170727
Recall	0.133472	0.094882
Precision	0.850931	0.850966
Accuracy	0.555175	0.515600

Confusion Matrix



6.6. Comparison to Spark MLLib

Here is the performance of the SVM implementation with respect to Spark MLLib on test dataset. In MLLib, we are not able to set the learning rate hyperparameter for SVM. And, it always predicted the negative label `not delayed` for all records. We also observed this behavior in our native implemention when we set the learning parameter to be greater than 0.1.

Metric	Native Implementation	Spark MLLib
Area Under ROC	0.538238	0.500000
Area Under PR	0.703687	0.525531
F1 score	0.170727	0.000000
Recall	0.094882	0.000000
Precision	0.850966	0.000000
Accuracy	0.515600	0.474469

Markdown

7. Conclusions

Our main evaluation metric was Area Under PR Curve due to the imbalanced nature of our dataset and also as we wanted to focus more on the positive cases than the negative cases. To recap the results from all the models implemented, please refer to the table below:

Test Metrics on 2019 heldout dataset

Model	Area Under PR	F1 score	Recall	Precision
Logistic Regression	0.362964	0.389682	0.591144	0.290633
Decision Trees	0.317968	0.383275	0.592206	0.283319
Gradient Boosted Trees	0.346511	0.375564	0.515503	0.295380
XGBoost	0.386705	0.400001	0.612218	0.297037

Our best model used XGBoost and had an Area Under PR Curve around 38%. Logistic Regression was the second-best model with Area Under PR Curve around 36%.

During the evaluation with 2019 held-out dataset, across our models, our precision was around 30% and area under PR curve was around 32-38%. One thing that we feel that might be impacting our results is that we did not do any dimensionality reduction such as PCA, to reduce the number of features in our model. That could be the reason why our model performance suffered from overfitting. Please refer to Section 4 for Area Under the PR Curve plots and the Confusion Matrices for various models.

We also evaluated our models against 2020 data and we noticed that the models performed worse than 2019 data evaluation. The precision is around 12-18% across our models with area under PR curve around 10-20%. We feel that this is due to the anamolous nature of the year 2020, there were other external factors like Covid-19 and related government measures like global shutdown etc that have not been represented in the features that we have considered. Please refer section 4.7.3 for the Area Under PR Curve plot and the Confusion Matrix plots. Please refer below table for 2020 results:

Test Metrics on 2020 dataset

Model	Area Under PR	F1 score	Recall	Precision
Logistic Regression	0.108296	0.197873	0.514534	0.122490
Decision Trees	0.136206	0.204086	0.373189	0.140446
Gradient Boosted Trees	0.170174	0.214192	0.265589	0.179462
XGBoost	0.195725	0.226878	0.386395	0.160583

Going back to the business case, these models are not used solely to predict flight delay but rather an accompanying datapoint on top of the specific domain knowledge held by stakeholders (flight control attendants, airline employees, etc). These models will complement their decision-making capabilities in making the announcement whether to delay a flight or not.

8. Course Concepts

1. *Custom Partitioning*: Initially using default paritioning, joining the airport and weather data took ~1.77 days to run. However, we figured that the default partitioning was set to 200 partitions. So we also set the `spark.sql.shuffle.partitions` to 1000 and the `spark.sql.files.minPartitionNum` to 1000 as we had 5 years of data. This helped improve the performance of our joins and the airlines and weather join complete in only 2.79 minutes
2. *One-hot encoding*: To include the categorical variables in our model, we had to convert these categorical variables into numerical form via one-hot encoding that did not reduce the amount of information in these variables. We pinpointed 7 categorical features to one-hot encode: `QUARTER`, `MONTH`, `DAY_OF_MONTH`, `DAY_OF_WEEK`, `OP_CARRIER_AIRLINE_ID`, and `DWND_BLK`. Even though the time-based variables were numerical, they were considered as categorical, not continuous. For example: quarter 1 vs quarter 2 is not incremental in terms of effects on our prediction. All we cared about is what value they hold. Thus, one-hot encoding was beneficial.
3. *GD - convex optimization/batch vs stochastic*: In the beginning of this course we learnt about how to distinguish between embarrassingly parallelizable and hard to parallelize jobs. For example deep learning is not an embarrassingly parallelizable computation, because you cannot compute each computation independently of other computations. This project helped us delve deeper into a better understanding of how to push computations/processes towards embarrassingly parallelizable, so we can use Spark and other parallel computation frameworks more efficiently. Additionally, in this class we also learnt about Batch Gradient Descent and Stochastic Gradient Descent. In the case of optimizing our Gradient Descent algorithm using parallelization we used Batch Gradient Descent instead of Stochastic Gradient Descent, because Stochastic Gradient Descent is not embarrassingly parallelizable. In our case we used analytical approaches as opposed to closed-form solutions, such as in SVM.
4. *Lazy evaluation*: One of the other concepts we learnt in this course is that Spark gains its computational optimization via lazy evaluation. The concept of lazy evaluation is that a computation is not done unless it is needed. This project helped us delve deeper into this technique for robust optimization.
5. *Caching (.cache() and .persist())*: Spark provides computation 100 times times faster than traditional Map-Reduce jobs. However if the jobs are designed to not reuse the repeating computations, we will see a downgrade in performance when we are dealing extremely large amounts of data. Hence, we may need to look at optimization techniques as to improve performance. There are two ways to do this. We can use .cache() and .persist(). We have used .persist() most widely. When we persist a dataset, each node stores its partitioned data in memory and reuses them in other actions on that dataset. And Spark's persisted data on nodes are fault-tolerant, meaning that if any partition of a dataset is lost, it will automatically be recomputed using the original transformations that created it.
6. *Scalability / time complexity*: One way we optimized on scalability and time complexity was to scale down on the data tables "before" performing the joins. We optimized the joins by filtering the data using different steps as outlined in the Joins section, so that we are dealing with only the relevant data.
7. *Broadcasting*: Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. When we implemented SVM, we used broadcasting for distributing model weights to all the variables
8. *Composite keys*: In Map Reduce models, composite keys can be used instead of the single keys for our parallel jobs. In our case we have used data frames. However, because we used multiple columns as our "keys", we technically used composite keys.
9. *Normalization*: Because the variables in our datasets have wildly different scales with some non-Gaussian distributions, we used Normalization in order to be able to compare all the different variables and be able to standardize them before training the model. Normalization essentially translates the data to the [0,1] range.
10. *Cross Validation*: Cross validation is a common approach to tune hyperparameters. However, since we are dealing with time-series data, it was important to choose the "expanding window" approach that avoided data leakage when training the models and tuning the hyperparameters. By using an expanding window, we benefited from testing the model on a quarter immediately following the training dataset.

```
1
```

Shift+Enter to run