Flight Delay Classification

PLAN

Business Problem:

We are assisting the Salt Lake City airport in predicting if certain weather conditions are likely to delay a flight. While it can be hard to predict storms, we have data on atmospheric conditions like air pressure, wind speed, and visibility that affect whether or not a flight is cleared for on-time departure.

For our business problem, we have assymetric loss. Delaying a flight is less costly than what could happen if a plane departed in unsafe weather conditions. For this analysis, we will say that delaying a flight that should have been cleared will lead to 50,000 dollars of costs from disrupting air traffic flow, while having an on-time flight that should have been delayed will lead to 500,000 dollars of costs from customer lawsuits over endangerment.

Ideal Dataset:

Since we are looking at weather information on the decision to delay a flight, we would want as many metrics of weather as possible. Our data has major components of weather: temperature, air pressure, wind speed, precipitation, and visibility. One shortcoming with our dataset is that delay information is split up into time intervals depending on how much a delay reason contributed to the total delay length. However, our data is fairly inconsistent with these time allotments. Ideally, we would want to only include delays that were caused by weather, but this is not feasible with our dataset.

BUILD

Loss Function:

In [448…

```
from sklearn.metrics import confusion_matrix
import numpy as np

def flight_loss(false_positives, false_negatives):
  endanger = false_negatives * 500000
```

```
    wasted = false_positives * 500000
    return endanger + wasted
```

Above is our piecewise loss function as described in the business problem. The 50,000 is our cost for false positives (delaying a flight that could have gone on-time), and the 500,000 is our cost for false negatives (letting a flight go on-time that should have been delayed). The "a" is the action we take and the "theta" is the state of the world determined by our cutoff value. When our action is "more than" the state of the world, we have a false positive and are charged 500,000, and when our action is "less than" the state of the world, we have a false negative and are charged 50,000. When our actions and the state of the world are the same, there are no costs.

Simulate Data:

In [449…

```
#Lets simulate some data
import numpy as np
import polars as pl
import seaborn.objects as so
import statsmodels.api as sm
import statsmodels.formula.api as smf
import bambi as bmb
import arviz as az

# Set randomization seed
rng = np.random.default_rng(42)

# Specify a function to simulate data
def sim_data(n, beta_0, beta_temp, beta_precipitation,
    beta_visibility, beta_airpress):
    # Simulate temperature using a normal distribution
    temp = rng.normal(54, 5, size=n)
    # Simulate precipitation using a binomial distribution
    precipitation = rng.binomial(1, 0.5, size=n)
    # Simulate visibility using a binomial distribution
    visibility = rng.binomial(1, 0.7, size=n)
    # Simulate air pressure using a binomial distribution
    airpress = rng.normal(54, 5, size=n)

    # Simulate the probability of a flight being delayed
    prob_y = (
        np.exp(beta_0 + beta_temp * temp + beta_precipitation *
               precipitation + beta_visibility *
               visibility + beta_airpress * airpress) /
        (1 + np.exp(beta_0 + beta_temp * temp + beta_precipitation
                    * precipitation + beta_visibility
                    * visibility + beta_airpress * airpress))
    )

    # Use prob_y to simulate the qualifed outcome variable
    delayed = rng.binomial(1, prob_y, size=n)

    # Return the output
    return delayed, temp, precipitation, visibility, airpress

# Call the function and save as an array
data_arr = sim_data(n = 500, beta_0 = 0.10, beta_temp = 0.05,
                    beta_precipitation = -.25,
                    beta_visibility = 0.09, beta_airpress = 0.08)

# Convert to a dataframe
data_df = pl.DataFrame(data_arr, schema = ['delayed', 'temp', 'precipitation',
                                           'visibility', 'airpress'])
```

In [450…

```
fr_fit = smf.glm(
  'delayed ~ temp + precipitation + visibility + airpress',
  data = data_df.to_pandas(),
  family = sm.families.Binomial()
).fit()
print(fr_fit.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                delayed   No. Observations:                  500
Model:                            GLM   Df Residuals:                      495
Model Family:                Binomial   Df Model:                            4
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -5.9458
Date:                Mon, 08 Dec 2025   Deviance:                       11.892
Time:                        20:08:04   Pearson chi2:                     141.
No. Iterations:                    25   Pseudo R-squ. (CS):           0.005058
Covariance Type:            nonrobust
=================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept        44.9995     3.3e+04      0.001      0.999   -6.47e+04    6.48e+04
temp             -0.0742      0.223     -0.333      0.739      -0.510       0.362
precipitation   -20.8640    2.07e+04     -0.001      0.999   -4.06e+04    4.06e+04
visibility      -20.3183    2.57e+04     -0.001      0.999   -5.05e+04    5.04e+04
airpress          0.1047      0.204      0.513      0.608      -0.295       0.505
=================================================================================
```

To check our ability to run a model, we simulated some data and fit it to a logistic regression model. Since we were able to recover all the parameters, we assume that if our simulated data matches parts of the ideal dataset we would be able to run a model on the actual data properly.

EXPLORE:

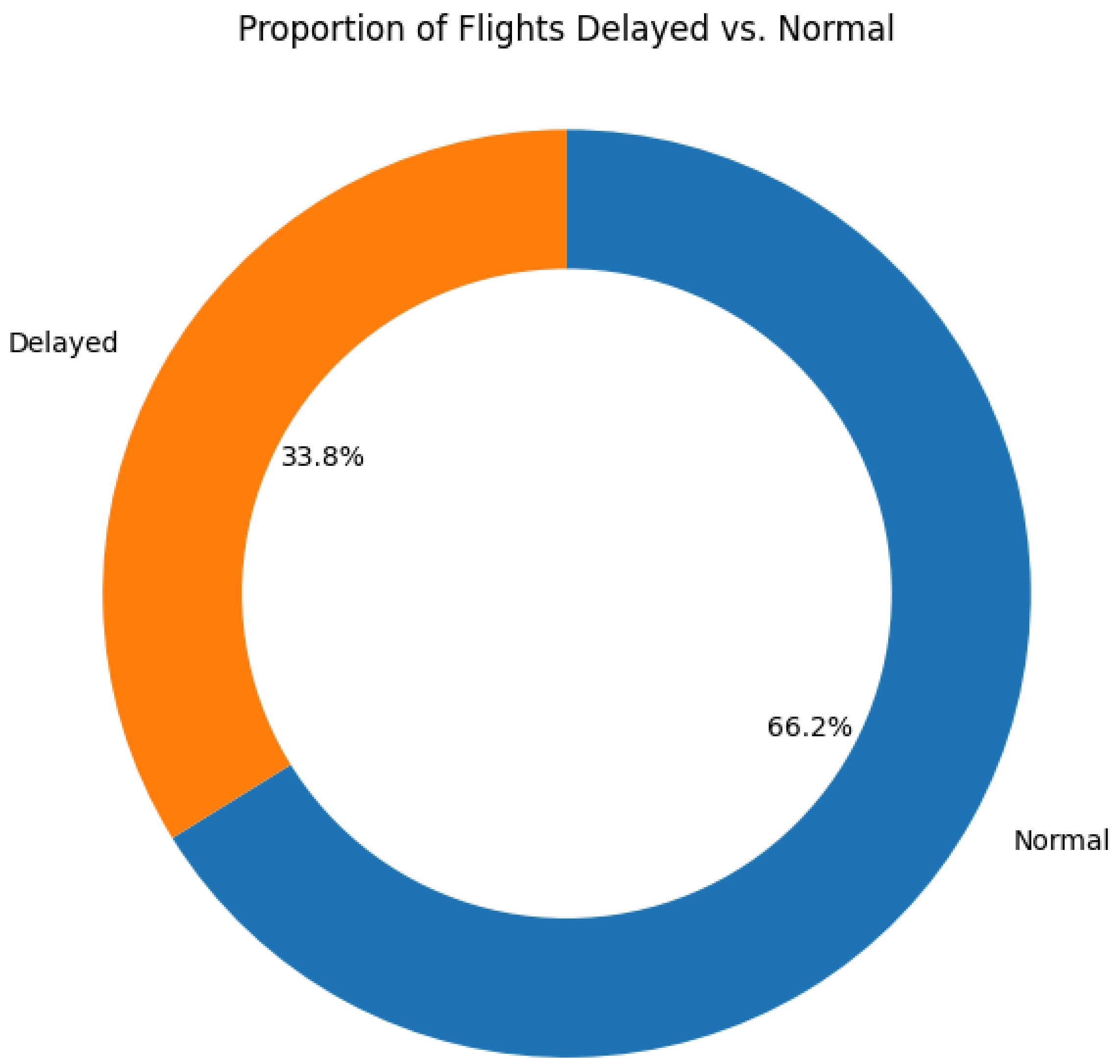
The dataset we have is merged from two sources: data for each flight came from the Bureau of Transportation Statistics of the United States, and the weather data came from the Iowa Environmental Mesonet at Iowa State University, which archives Automated Surface Observing System (ASOS) data from airports. Our dataset only contains observations from the Salt Lake City airport. It spans from 1/1/2021 to 9/30/2023.

In [451…

```python
#Import SLC.csv as a pandas dataframe
import pandas as pd
df = pd.read_csv('SLC.csv')

#Map Delayed and Cancelled 'TRUE' and 'FALSE' to 1 and 0
df['Cancelled'] = df['Cancelled'].map({True: 1, False: 0})
df['Delayed'] = df['Delayed'].map({True: 1, False: 0})
```

In [452…

```python
#Create a donut chart with the proportion of delays
# and on-time flights
import matplotlib.pyplot as plt

# Create a column for flight status
df['FlightStatus'] = 'Normal'
df.loc[df['Delayed'] == 1, 'FlightStatus'] = 'Delayed'

# Get the counts for each status
status_counts = df['FlightStatus'].value_counts()

# Create the donut chart using Matplotlib
fig, ax = plt.subplots(figsize=(6, 6))
wedges, texts, autotexts = ax.pie(
    status_counts.values,
    labels=status_counts.index,
    autopct='%1.1f%%',
    startangle=90,
    counterclock=False
)
# Draw a white circle in the center to create a donut
centre_circle = plt.Circle((0, 0), 0.70, fc='white')
fig.gca().add_artist(centre_circle)
ax.set_title("Proportion of Flights Delayed vs. Normal")
ax.axis('equal')
plt.tight_layout()
plt.show()
```

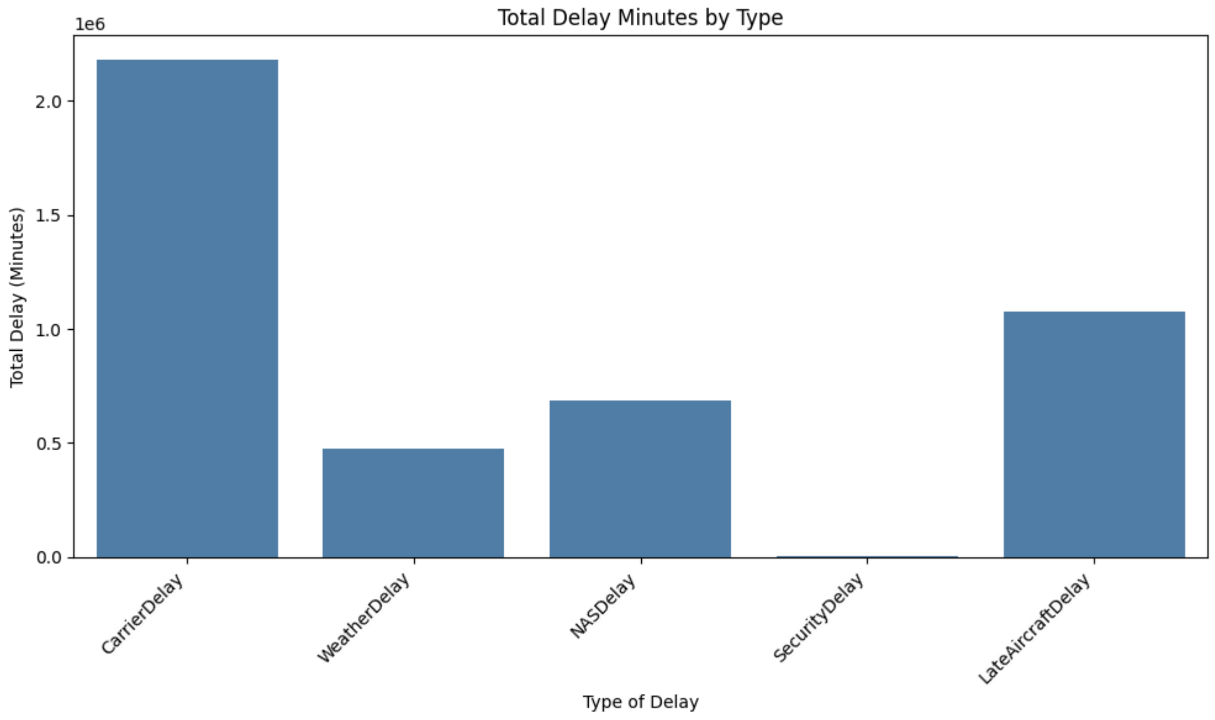


This first chart shows the percentages of flights that leave on time or are delayed. We see that roughly one-third of flights are delayed and two-thirds of flights leave on-time. Our model needs to obtain an accuracy of 66.2% or more to beat a random guess using these percentages.

In [453…

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Identify columns that end with 'Delay'
delay_columns = [col for col in df.columns if col.endswith('Delay')]

# Sum the values for each identified column
delay_sums = df[delay_columns].sum()

# Convert the sums to a DataFrame for plotting
delay_sums_df = delay_sums.reset_index()
delay_sums_df.columns = ['DelayType', 'TotalDelay']

# Create a bar chart with Seaborn/Matplotlib
```

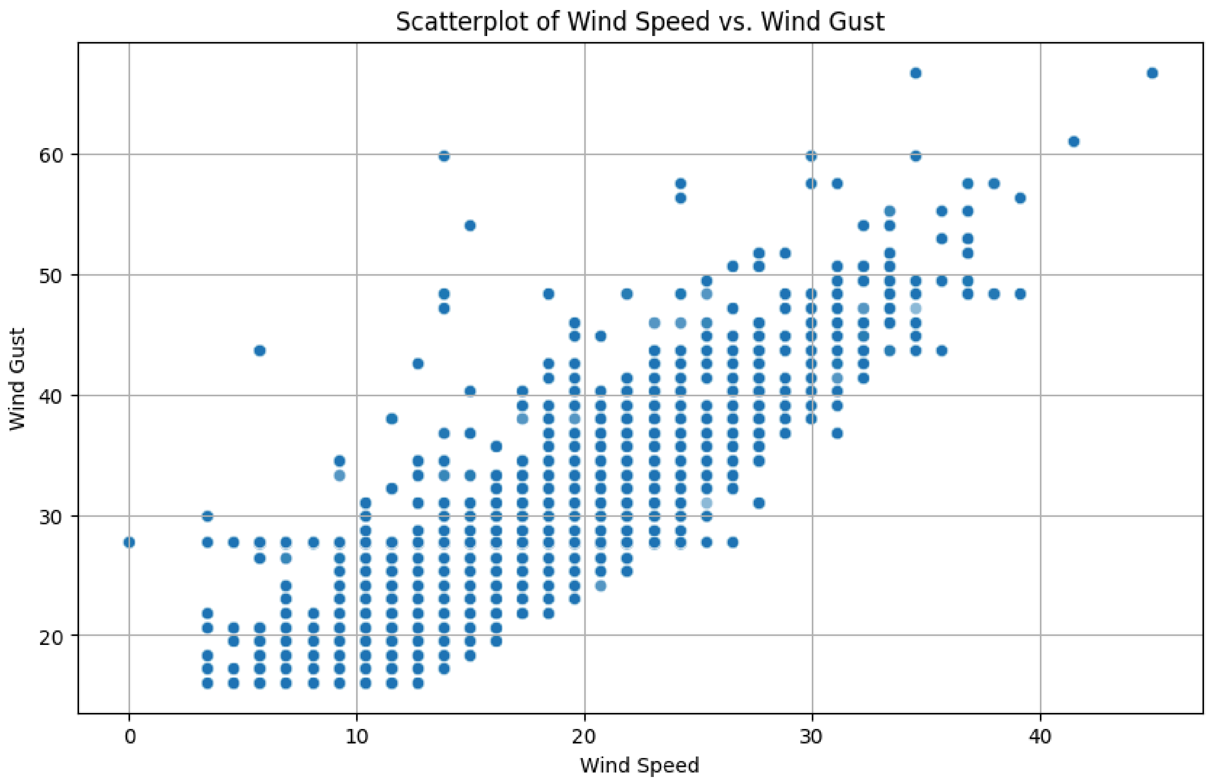
```
plt.figure(figsize=(10, 6))
sns.barplot(data=delay_sums_df, x='DelayType', y='TotalDelay', color='steelblue')
plt.title('Total Delay Minutes by Type')
plt.xlabel('Type of Delay')
plt.ylabel('Total Delay (Minutes)')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



The second chart shows us how much delay time is attributed to certain causes. We see that security concerns contribute very little to delay time, while carriers and late aircraft contribute most to the delay time. Relative to these causes, local weather plays a small role in delay time.

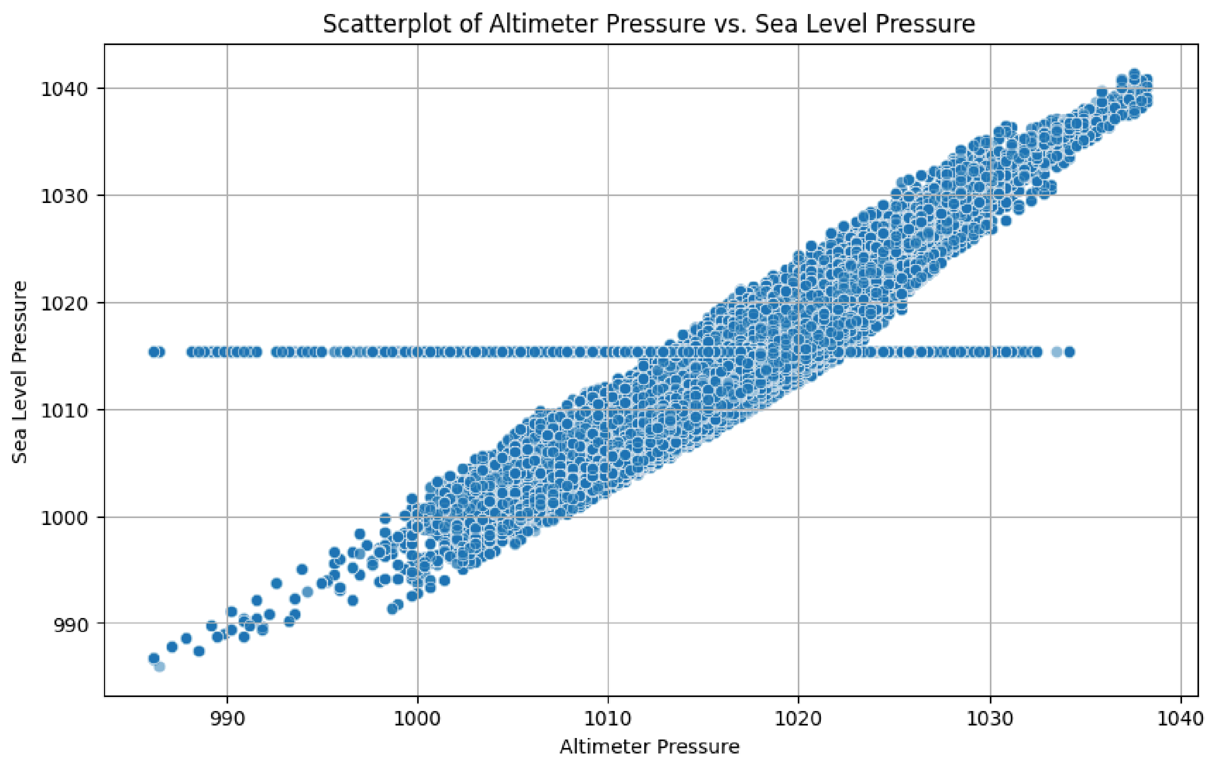
In [454…

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Wind_Speed', y='Wind_Gust', alpha=0.5)
plt.title('Scatterplot of Wind Speed vs. Wind Gust')
plt.xlabel('Wind Speed')
plt.ylabel('Wind Gust')
plt.grid(True)
plt.show()
```



In [455…

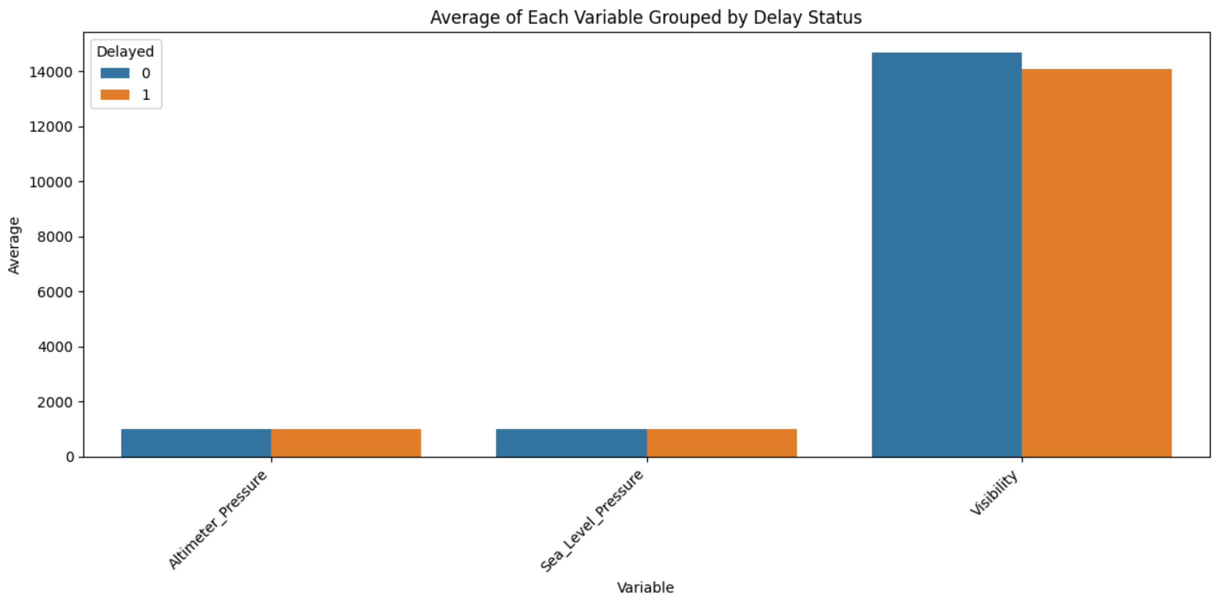
```python
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Altimeter_Pressure',
    y='Sea_Level_Pressure', alpha=0.5)
plt.title('Scatterplot of Altimeter Pressure vs. Sea Level Pressure')
plt.xlabel('Altimeter Pressure')
plt.ylabel('Sea Level Pressure')
plt.grid(True)
plt.show()
```



These two scatterplots show the only positive correlations that we could find in our dataset. An increase of wind speed is correlated with an increase in wind gust, and an increase of altimeter pressure is correlated with an increase in sea level pressure.

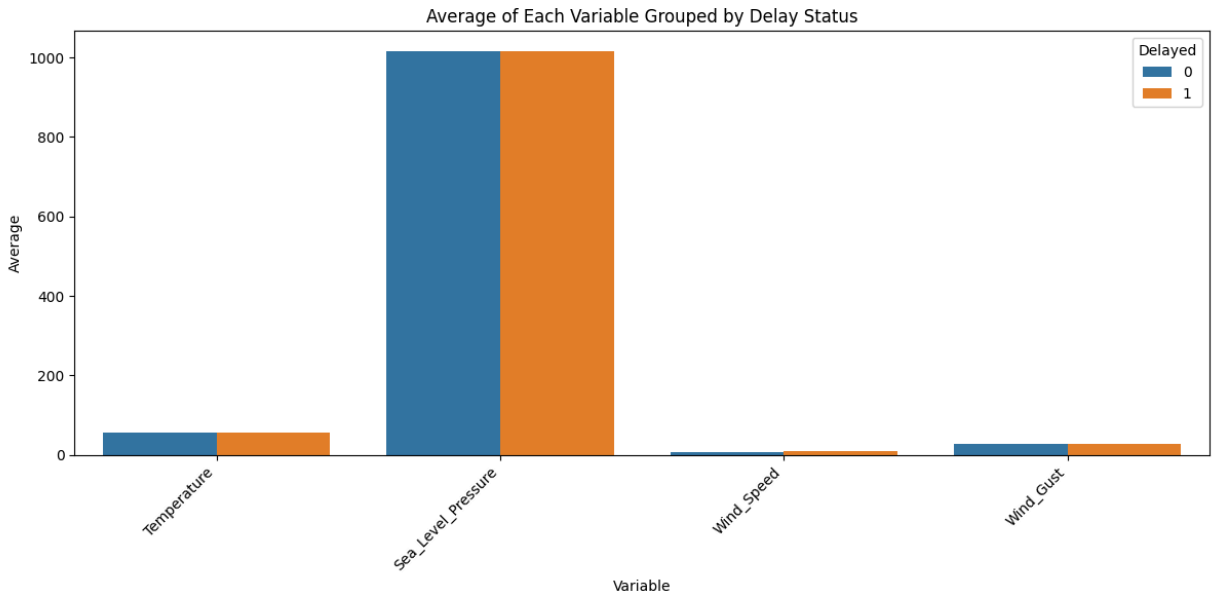
In [456…

```
import matplotlib.pyplot as plt
import seaborn as sns

# Compute averages for all numeric columns grouped by Delayed
avg_df = df.groupby("Delayed").mean(numeric_only=True).reset_index()
avg_df = avg_df.drop(columns=[#'Visibility', 'Altimeter_Pressure',
                              # 'Sea_Level_Pressure',
                              'Cancelled', 'DepDelayMinutes',
                              'CarrierDelay', 'WeatherDelay',
                              'SecurityDelay', 'LateAircraftDelay',
                              'NASDelay', 'Feels_Like_Temperature',
                              'Temperature', 'Wind_Speed', 'Wind_Gust',
                              'Precipitation',
                              'Ice_Accretion_3hr'
                              ])

# Melt (reshape) to long format for easy plotting
avg_melted = avg_df.melt(id_vars="Delayed",
    var_name="Variable", value_name="Average")

# Ensure Delayed is categorical so Seaborn groups correctly
avg_melted["Delayed"] = avg_melted["Delayed"].astype(str)

# Seaborn grouped bar chart
plt.figure(figsize=(12, 6))
sns.barplot(data=avg_melted, x="Variable", y="Average", hue="Delayed")
plt.title("Average of Each Variable Grouped by Delay Status")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



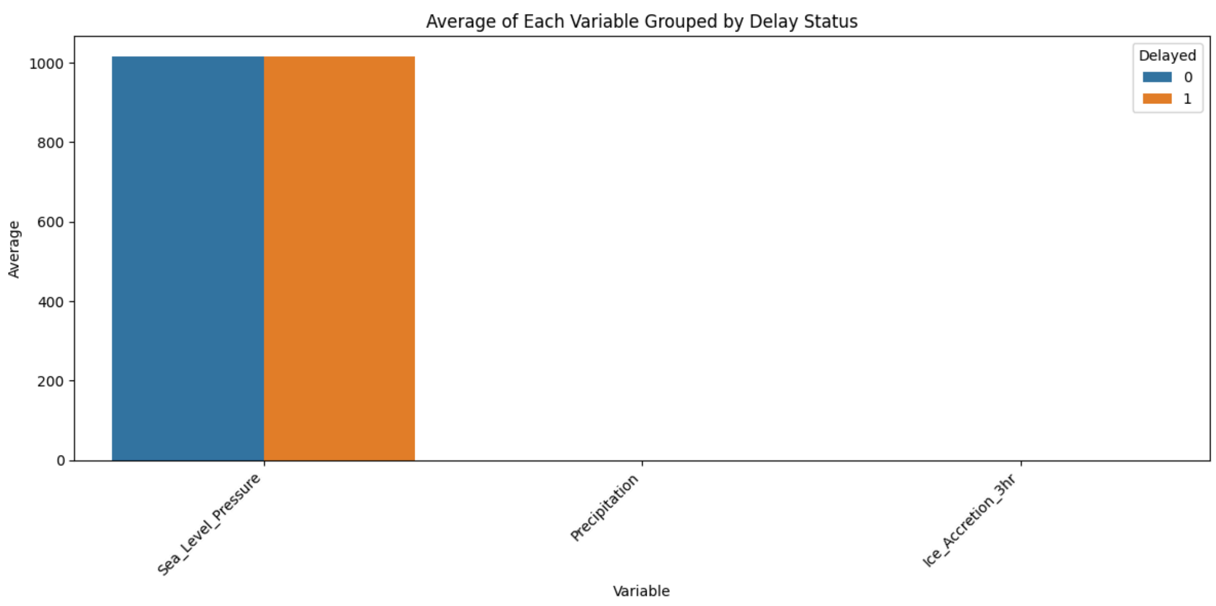
In [457…

```
import matplotlib.pyplot as plt
import seaborn as sns

# Compute averages for all numeric columns grouped by Delayed
avg_df = df.groupby("Delayed").mean(numeric_only=True).reset_index()
avg_df = avg_df.drop(columns=['Visibility', 'Altimeter_Pressure',
                              # 'Sea_Level_Pressure',
                              'Cancelled', 'DepDelayMinutes',
                              'CarrierDelay', 'WeatherDelay',
                              'SecurityDelay', 'LateAircraftDelay',
                              'NASDelay', 'Feels_Like_Temperature',
                              #'Temperature', 'Wind_Speed', 'Wind_Gust',
                              'Precipitation',
                              'Ice_Accretion_3hr'
                              ])

# Melt (reshape) to long format for easy plotting
avg_melted = avg_df.melt(id_vars="Delayed",
     var_name="Variable", value_name="Average")

# Ensure Delayed is categorical so Seaborn groups correctly
avg_melted["Delayed"] = avg_melted["Delayed"].astype(str)

# Seaborn grouped bar chart
plt.figure(figsize=(12, 6))
sns.barplot(data=avg_melted, x="Variable", y="Average", hue="Delayed")
plt.title("Average of Each Variable Grouped by Delay Status")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



In [458…

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Compute averages for all numeric columns grouped by Delayed
avg_df = df.groupby("Delayed").mean(numeric_only=True).reset_index()
avg_df = avg_df.drop(columns=['Visibility', 'Altimeter_Pressure',
                              # 'Sea_Level_Pressure',
                              'Cancelled', 'DepDelayMinutes',
                              'CarrierDelay', 'WeatherDelay',
                              'SecurityDelay', 'LateAircraftDelay',
                              'NASDelay', 'Feels_Like_Temperature',
                              'Temperature', 'Wind_Speed', 'Wind_Gust',
                              #'Precipitation',
                              #'Ice_Accretion_3hr'
                              ])

# Melt (reshape) to long format for easy plotting
avg_melted = avg_df.melt(id_vars="Delayed",
    var_name="Variable", value_name="Average")

# Ensure Delayed is categorical so Seaborn groups correctly
avg_melted["Delayed"] = avg_melted["Delayed"].astype(str)

# Seaborn grouped bar chart
plt.figure(figsize=(12, 6))
sns.barplot(data=avg_melted, x="Variable", y="Average", hue="Delayed")
plt.title("Average of Each Variable Grouped by Delay Status")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



These charts show the differences in average values for each variable, grouped by whether the flight was delayed or not. Altimeter pressure and sea level pressure have little change in averages between the delayed and on-time groups. However, we do see a lower average temperature and visibility in the delayed flight set, as well as higher precipitation. From our charts, there is also a small increase in wind speed and wind gust in the delayed dataset. To fully understand the differences in means, we would need to use the standard deviation of each variable and check for statistical significance.

Data Cleaning

In [459…

```python
#Remove all columns related to delay information besides the binary indicator
df = df.drop(columns=['DepDelayMinutes', 'CarrierDelay', 'WeatherDelay',
    'SecurityDelay', 'LateAircraftDelay', 'NASDelay'])

#Remove other unnecessary columns
df = df.drop(columns=['Time', 'Dest', 'Carrier', 'Origin', 'Cancelled',
    'CancellationReason', 'Feels_Like_Temperature'])
```

In [460…

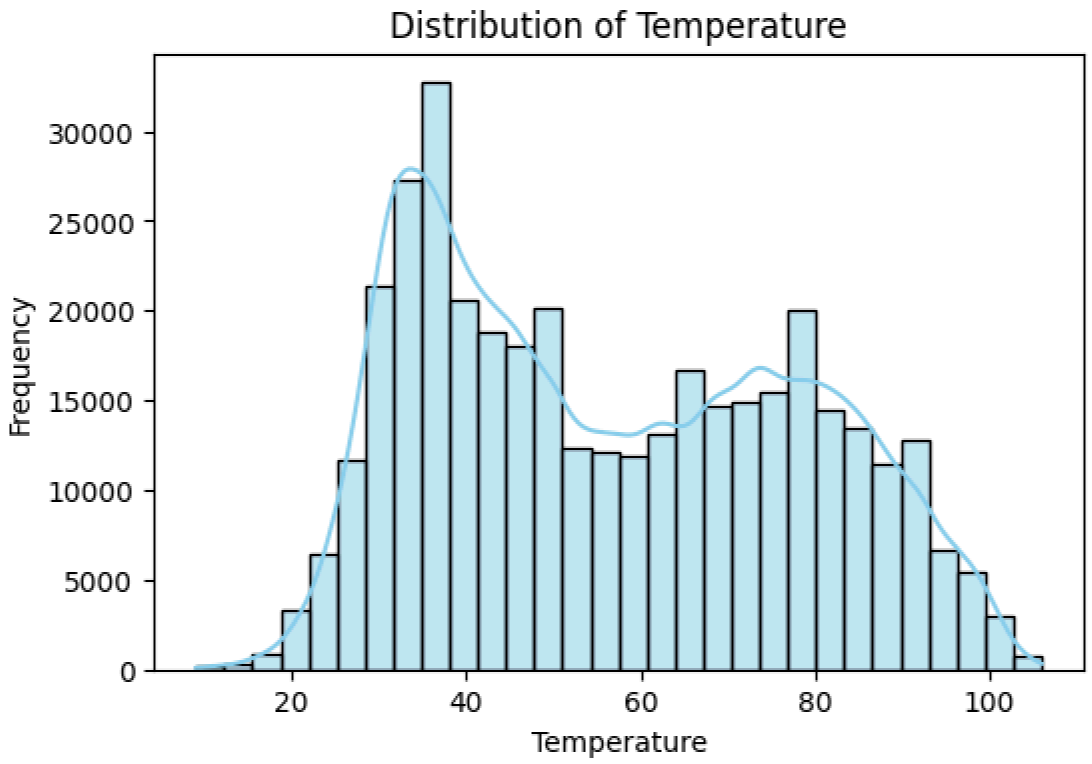
```python
# Dropping redundant column that matches our column we want to predict
df = df.drop('FlightStatus', axis = 1)
```

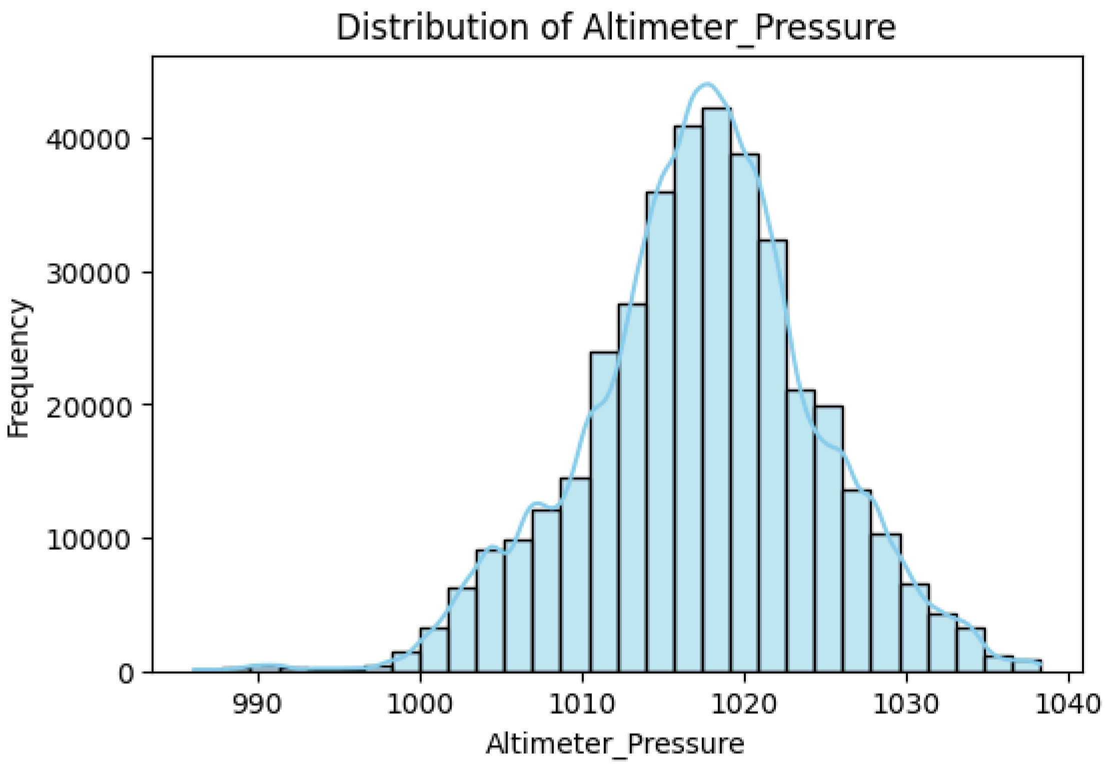
Since we are only looking at delayed flights, we dropped all columns relating to flight cancellations. We also dropped any delays that were not due to the weather before making our mean comparison charts, as they would throw off our estimates based on weather patterns.

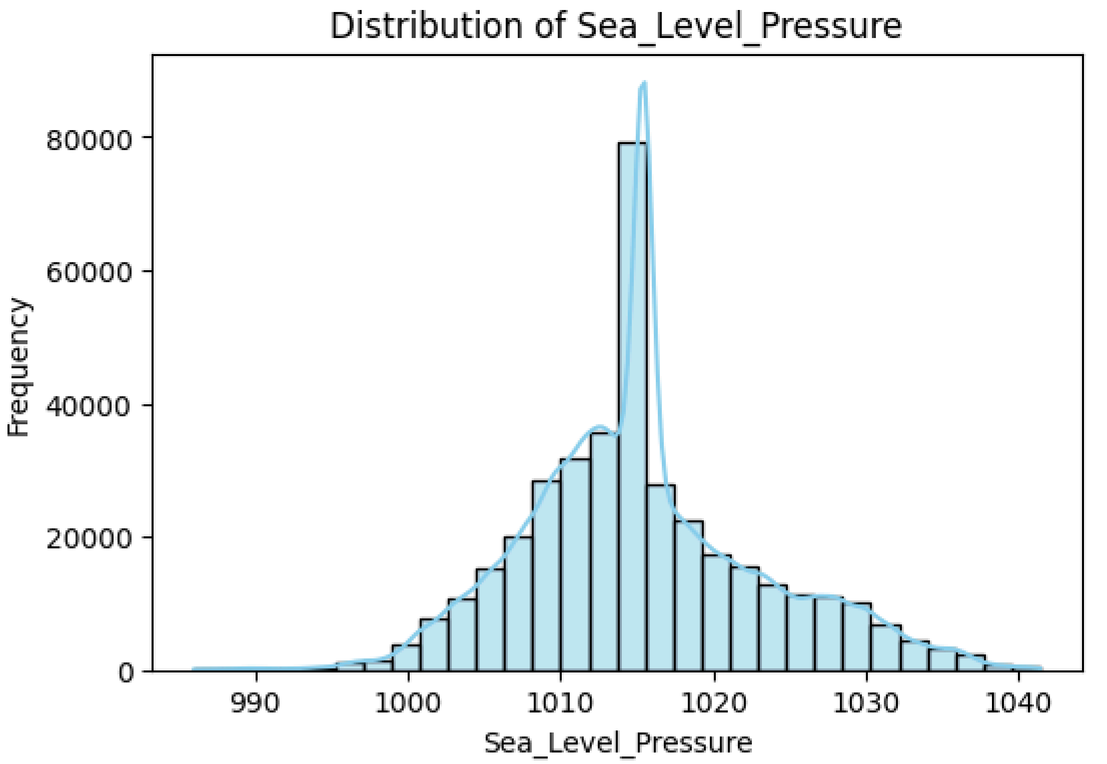
In [461…

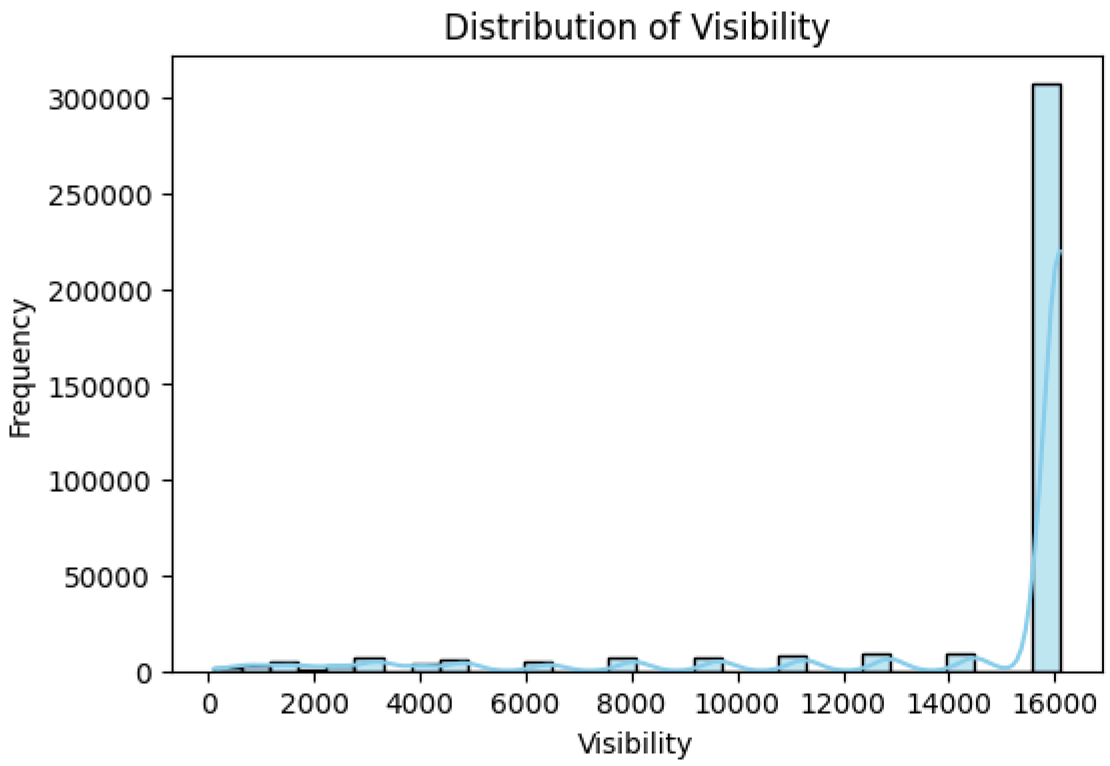
```python
# Create a histogram for each quantitative variable to check for normality
import seaborn as sns
import matplotlib.pyplot as plt
quant_vars = ['Temperature', 'Altimeter_Pressure',
    'Sea_Level_Pressure', 'Visibility',
    'Wind_Speed', 'Wind_Gust', 'Precipitation', 'Ice_Accretion_3hr']
plt.figure(figsize=(15, 4))
for var in quant_vars:
    plt.figure(figsize=(6, 4))
    sns.histplot(df[var], kde=True, bins=30, color='skyblue')
    plt.title(f'Distribution of {var}')
    plt.xlabel(var)
    plt.ylabel('Frequency')
    plt.show()
```

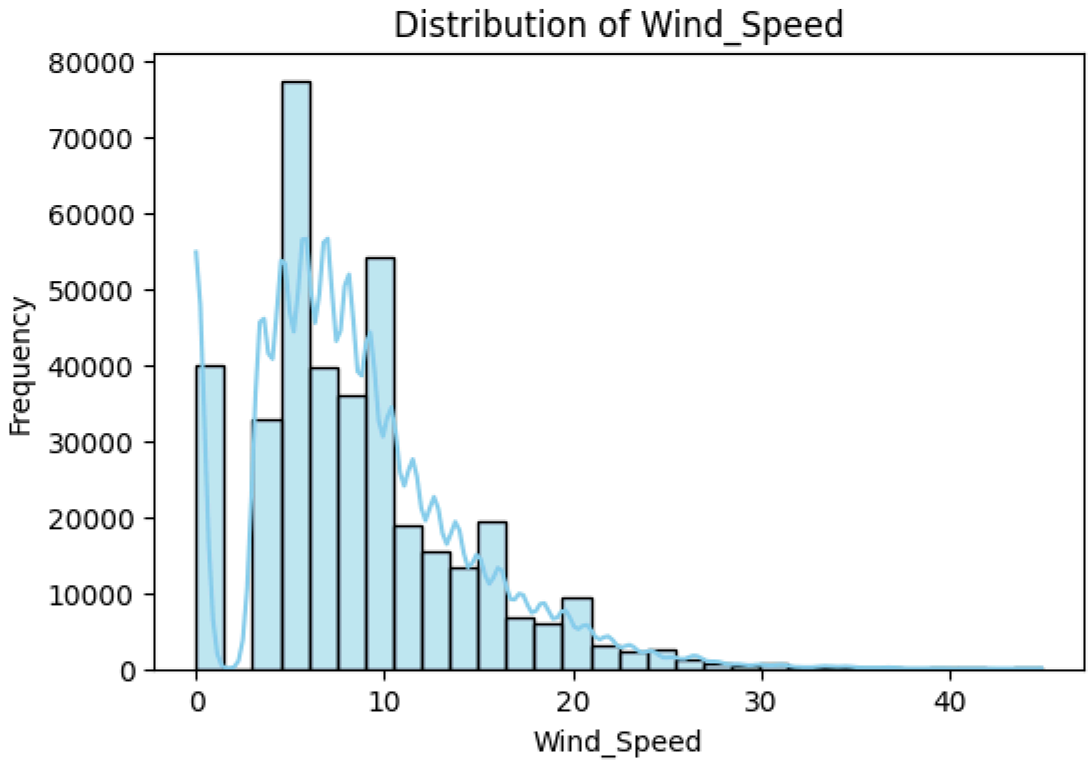
```
<Figure size 1500x400 with 0 Axes>
```

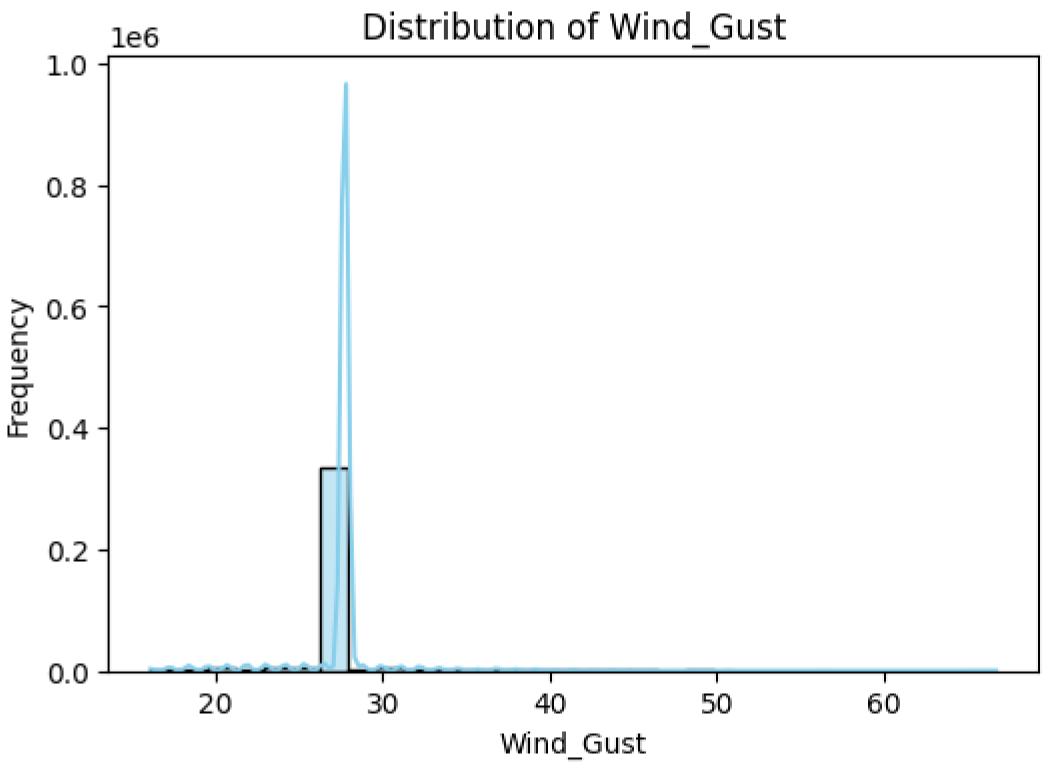


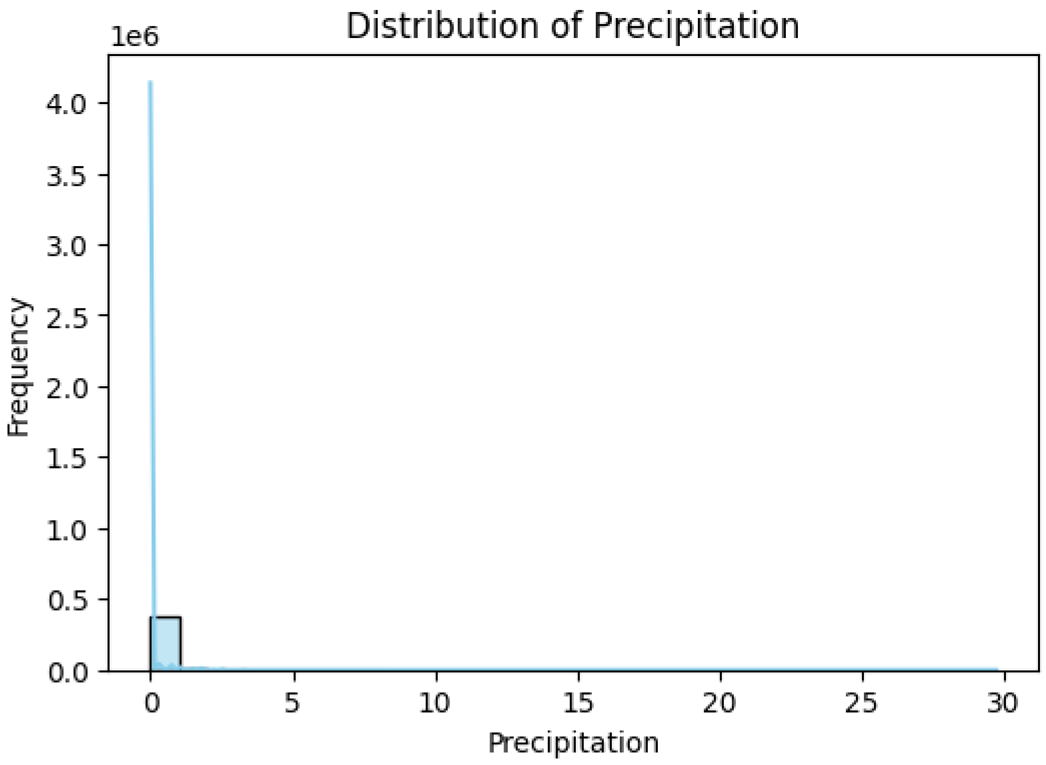

Distribution of Altimeter_Pressure
Frequency
0
10000
20000
30000
40000
990
1000
1010
1020
1030
1040
Altimeter_Pressure

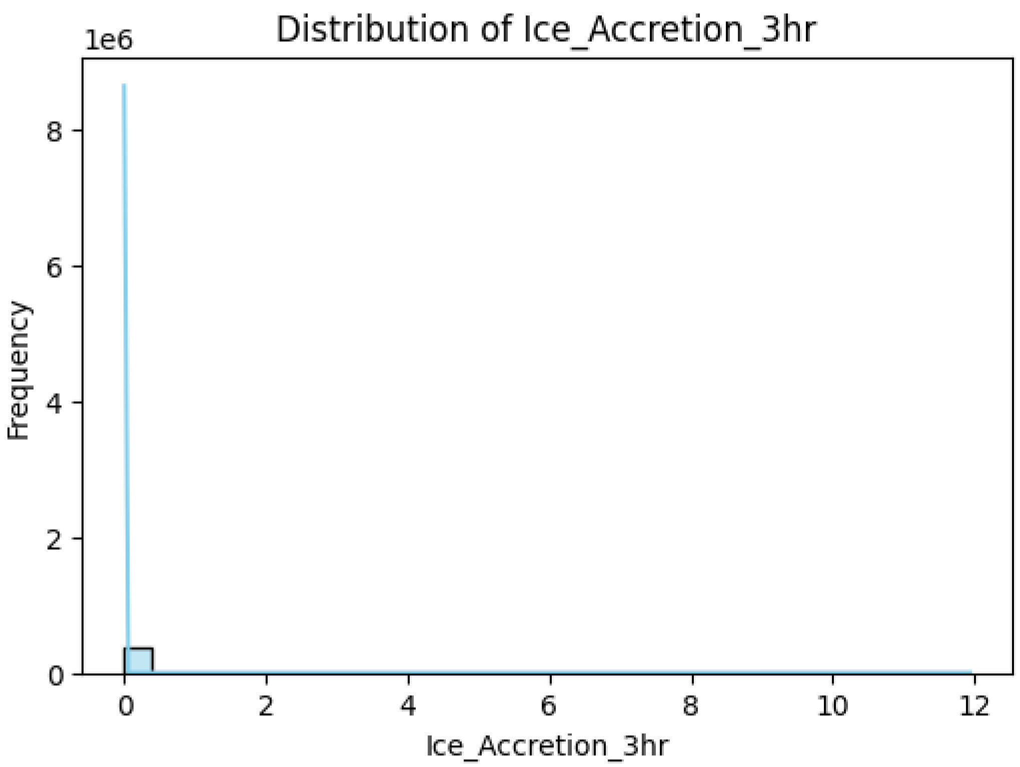

Distribution of Sea_Level_Pressure
Frequency
0
20000
40000
60000
80000
990
1000
1010
1020
1030
1040
Sea_Level_Pressure


Distribution of Visibility
Frequency
300000
250000
200000
150000
100000
50000
0
0
2000
4000
6000
8000
10000
12000
14000
16000
Visibility


Distribution of Wind_Speed
Frequency
80000
70000
60000
50000
40000
30000
20000
10000
0
0
10
20
30
40
Wind_Speed


Distribution of Wind_Gust
1e6
1.0
0.8
0.6
0.4
0.2
0.0
Frequency
20
30
40
50
60
Wind_Gust


Distribution of Precipitation
1e6
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Frequency
0
5
10
15
20
25
30
Precipitation



We assessed the normality of each predictor in our dataset. Visibilitiy, precipitation, and ice accretion have the highest amounts of skew. However, since we are running a principal components analysis later, we will not log transform any variables.

Logistic Regression

We want to start fitting our data to a model. We are going to begin with a Logistic Regression Model.

RECONCILE:

Before running our model, we are going to make sure our data fit all the assumptions of a generalized linear model.

Validity - We assume the data to be valid. It was collected from the U.S. government's air traffic statistics and airport weather tracking units. All of the major weather metrics are included in our dataset, as well as basic information about each flight, which should give us all the variables of our ideal data.

Representativeness - The data we have is representative, as it records every flight out of Salt Lake City in the defined period. To check for influential points, we will fit a linear regression model and check for standard residuals greater than 2.

In [462…
```
df.head()
```

Out[462...

	Delayed	Temperature	Altimeter_Pressure	Sea_Level_Pressure	Visibility	Wind_Speed	W
0	1	35.0	1023.03	1024.0	16093.40	11.51	
1	1	27.0	1028.78	1031.1	16093.40	0.00	
2	0	25.0	1025.40	1027.5	14484.06	9.21	
3	0	26.1	1026.08	1015.4	12874.72	0.00	
4	0	27.0	1026.08	1028.5	16093.40	4.60	

In [463...

```python
import statsmodels.api as sm
import statsmodels.formula.api as smf

model_formula = '''Delayed ~ Temperature +
    Altimeter_Pressure + Sea_Level_Pressure +
    Visibility + Wind_Speed + Wind_Gust + Precipitation + Ice_Accretion_3hr'''
logit_model = smf.glm(formula=model_formula, data=df,
    family=sm.families.Binomial()).fit()

influence = logit_model.get_influence()
summary = influence.summary_frame()

# Extract standard residuals
standard_resid = summary['standard_resid']

# Filter for standard residuals greater than 2
residuals_greater_than_2 = standard_resid[standard_resid > 2]

print("Standard Residuals greater than 2:")
print(residuals_greater_than_2)
```

```
Standard Residuals greater than 2:
254960    2.009518
264983    2.009518
290685    2.169431
Name: standard_resid, dtype: float64
```

In [464...

```python
print(df.iloc[254960])
print(df.iloc[264983])
print(df.iloc[290685])
```

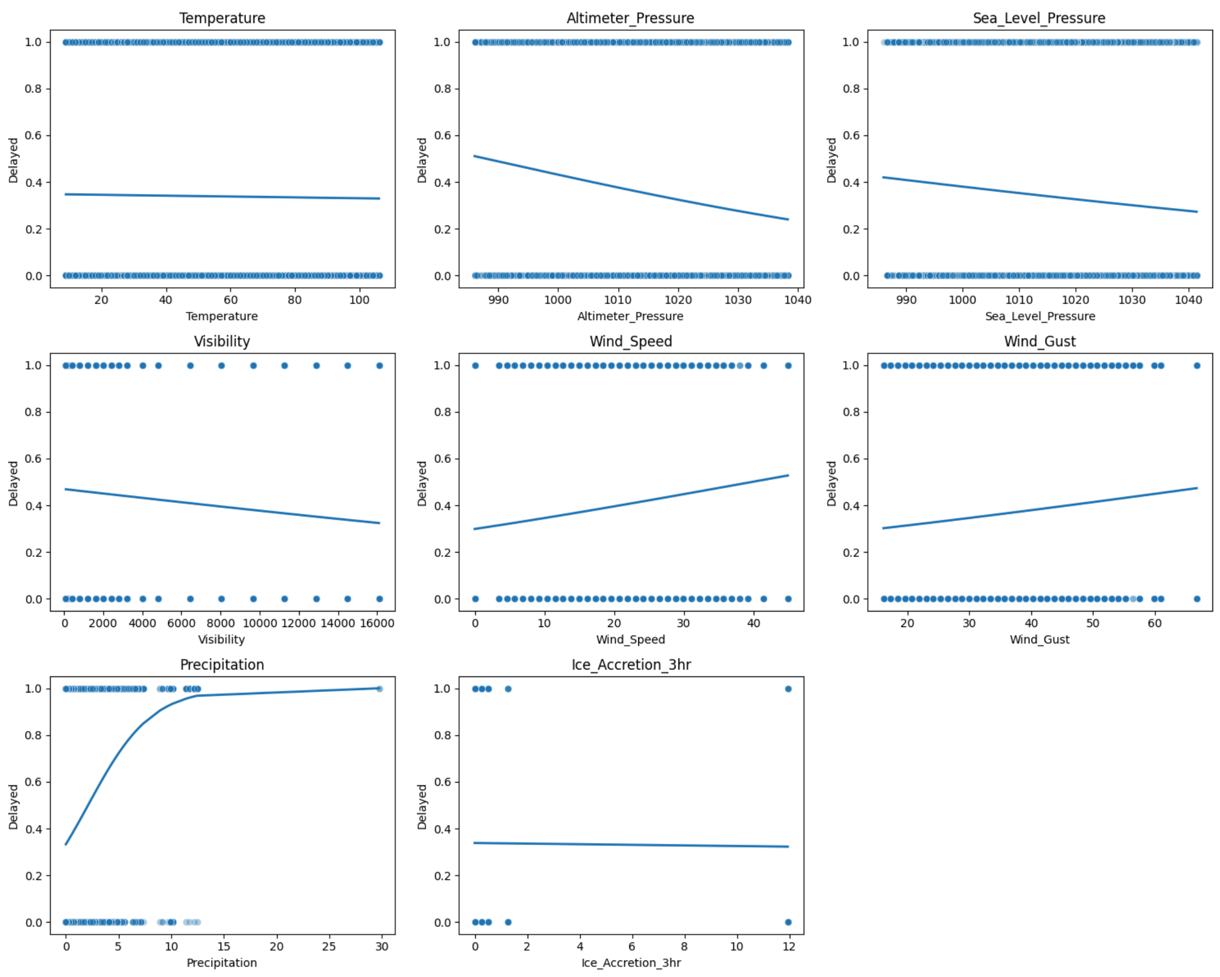
```
Delayed                  1.00
Temperature             28.00
Altimeter_Pressure    1034.20
Sea_Level_Pressure    1015.40
Visibility           11265.38
Wind_Speed               3.45
Wind_Gust               27.73
Precipitation            0.00
Ice_Accretion_3hr        0.00
Name: 254960, dtype: float64
Delayed                  1.00
Temperature             28.00
Altimeter_Pressure    1034.20
Sea_Level_Pressure    1015.40
Visibility           11265.38
Wind_Speed               3.45
Wind_Gust               27.73
Precipitation            0.00
Ice_Accretion_3hr        0.00
Name: 264983, dtype: float64
Delayed                  1.00
Temperature             35.60
Altimeter_Pressure    1034.20
Sea_Level_Pressure    1015.40
Visibility           16093.40
Wind_Speed               0.00
Wind_Gust               27.73
Precipitation            0.00
Ice_Accretion_3hr        0.00
Name: 290685, dtype: float64
```

Even though there are three values with standard residuals greater than 2, we assume that these are naturally-occuring influential points and should not be dropped from our dataset.

Independence - We did not observe any clustering within the dataset. Since we removed our date column, there is no time-series analysis occuring. Each row contains a unique flight, and we have randomly sampled to get our training and test datasets.

Linearity - To check our linearity, we will create a scatterplot of the log-odds and Delayed.

In [ ]:
```python
#Create a scatterplot of log-odds for every variable and Delayed
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
import numpy as np

cols = 3
rows = (len(quant_vars) // cols) + (len(quant_vars) % cols > 0)
plt.figure(figsize=(cols * 5, rows * 4))

for i, col in enumerate(quant_vars):
    plt.subplot(rows, cols, i + 1)

    # scatter of binary outcome
    sns.scatterplot(
        x=df[col],
        y=df['Delayed'],
        alpha=0.35
    )

    x = df[col]
    y = df['Delayed']

    # remove missing
    mask = ~(x.isna() | y.isna())
    x_clean = x[mask]
    y_clean = y[mask]

    # fit logistic model
    X = sm.add_constant(x_clean)
    model = sm.Logit(y_clean, X).fit(disp=0)

    # sorted prediction line
    sort_idx = np.argsort(x_clean)
    x_sorted = x_clean.iloc[sort_idx]
    pred_sorted = model.predict(X.iloc[sort_idx])

    # plot the smoothed logistic curve
    sns.lineplot(
        x=x_sorted,
        y=pred_sorted,
        linewidth=2
    )

    # titles & formatting
    plt.title(col)
    plt.ylim(-0.05, 1.05)

plt.tight_layout()
plt.show()
plt.close('all')
```



Looking at our scatterplots of delayed and the log-odds of each variable, we can see that they are all monotone, meaning that these predictors will work well with our logistic regression model. From this diagnostic, we expect that delays are associated with a higher level of wind and precipitation, as well as lower levels of pressure and visibility. Temperature and ice accretion seem to have slight downward trends, but they are less profound than the other variables.

Multicollinearity - From an inital look at our variables, we assume that altimeter pressure and sea pressure level are coordinated, as well as wind gust and wind speed. To check for multicollinearity, we will compute variance inflation factors for all our predictors.

In [466…

```python
# Compute VIF using all quantitative predictors
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm
import pandas as pd

predictors = df[quant_vars]
X = sm.add_constant(predictors)

# Calculate VIF for each predictor
vif_data = pd.DataFrame()
vif_data["feature"] = X.columns
vif_data["VIF"] = [variance_inflation_factor(X.values, i)
```

```
    for i in range(X.shape[1])]

print(vif_data)
```

```
             feature           VIF
0              const  27732.782996
1        Temperature      2.366523
2  Altimeter_Pressure     6.303433
3  Sea_Level_Pressure     7.639463
4         Visibility      1.326535
5         Wind_Speed      1.323751
6          Wind_Gust      1.117202
7      Precipitation      1.106979
8   Ice_Accretion_3hr     1.000387
```

In [467…

```
# Recompute VIF without Sea Level Pressure
import pandas as pd
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm

predictors = df[quant_vars].copy()

# Drop 'Sea_Level_Pressure' as requested
predictors = predictors.drop('Sea_Level_Pressure', axis=1)

X2 = sm.add_constant(predictors)

# Calculate VIF for each predictor
vif_data_updated = pd.DataFrame()
vif_data_updated["feature"] = X2.columns
vif_data_updated["VIF"] = [variance_inflation_factor(X2.values, i)
    for i in range(X2.shape[1])]

print(vif_data_updated)
```

```
             feature           VIF
0              const  25851.939444
1        Temperature      1.242151
2  Altimeter_Pressure     1.246023
3         Visibility      1.322280
4         Wind_Speed      1.321059
5          Wind_Gust      1.116729
6      Precipitation      1.106883
7   Ice_Accretion_3hr     1.000386
```

Our VIF analysis looks good. The only variables that are highly coordinated are altimeter pressure and sea level pressure, which were giving VIFs of 6.3 and 7.6 respectively. After removing Sea Level Pressure from the analysis, all our variables are under a VIF of 2. Since we will be trying out forms of penalized regression that reduce the effect of multicollinearity, we will keep all our predictors in the dataset.

FIT:

We are going to start fitting the model with some feature engineering to make sure our data is ready for the model.

In [468…

```
from sklearn.model_selection import train_test_split
```

In [469…

```
# Running code to reduce our dataset down to 5000 to reduce computation time
# We want to make sure our distribution is saved so we will use stratify.
_, df_sampled = train_test_split(
        df,
        test_size=5000,
        stratify=df['Delayed'],
        random_state=42
    )

print(f"Original DataFrame shape: {df.shape}")
print(f"Sampled DataFrame shape: {df_sampled.shape}")
print("Distribution of Delayed in original df:")
print(df['Delayed'].value_counts(normalize=True))
print("\nDistribution of Delayed in sampled df:")
print(df_sampled['Delayed'].value_counts(normalize=True))
```

```
Original DataFrame shape: (380735, 9)
Sampled DataFrame shape: (5000, 9)
Distribution of Delayed in original df:
Delayed
0    0.66167
1    0.33833
Name: proportion, dtype: float64

Distribution of Delayed in sampled df:
Delayed
0    0.6616
1    0.3384
Name: proportion, dtype: float64
```

In [470…

```
# Separate features (X) and target (y)
X = df_sampled.drop('Delayed', axis=1)
y = df_sampled['Delayed']

# Splitting the data for hyper parameter tuning
X_train, X_test, y_train, y_test = train_test_split(X,
        y,
        test_size=0.3,
        stratify=y,
        random_state=42)

# Combine y_train and X_train
flights_train = X_train.copy()
flights_train['Delayed'] = y_train
```

We have reduced the number of rows in our dataset to account for long computation times. It keeps the original distribution of our target variable.

In [471…

```
from sklearn.model_selection import StratifiedKFold
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cutoffs = [0.3, .325, 0.35, 0.375, 0.4, 0.425, 0.45, 0.475, 0.5,
    0.525, 0.55, 0.575, 0.6, 0.625, 0.65, 0.675, 0.7, 0.725, 0.75]
```

In [472…

```
y_train_r = y_train.values.ravel()
```

In [473…

```
import warnings
warnings.filterwarnings("ignore", message="overflow encountered in exp")
warnings.filterwarnings("ignore", category=Warning)
```

In [474…

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.linear_model import LogisticRegression
import numpy as np

results = []

# running the loop
for c in cutoffs:
    losses = []  # store loss for this cutoff

    for train_idx, val_idx in kf.split(X_train, y_train_r):
        # Split into train/validation
        X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
        y_tr, y_val = y_train_r[train_idx], y_train_r[val_idx]

        # Fit Logistic Regression model
        model = LogisticRegression(max_iter=1000).fit(X_tr, y_tr)

        # Predict probabilities for the positive class
        y_prob = model.predict_proba(X_val)[:, 1]

        # Apply cutoff
        y_pred = (y_prob >= c).astype(int)

        # Confusion matrix
        conf_matrix = confusion_matrix(y_val, y_pred)

        # Extract FP and FN
        false_positives = conf_matrix[0, 1]
        false_negatives = conf_matrix[1, 0]

        # Compute loss
        loss = flight_loss(false_positives, false_negatives)
        losses.append(loss)

    # store results for this cutoff
    results.append({
        'cutoff': c,
        'loss': np.mean(losses),
    })

# print clean output
```

```python
for result in results:
    print(f"Cutoff: {result['cutoff']}, Flight Loss: {result['loss']}")
```

```
Cutoff: 0.3, Flight Loss: 206300000.0
Cutoff: 0.325, Flight Loss: 170400000.0
Cutoff: 0.35, Flight Loss: 139900000.0
Cutoff: 0.375, Flight Loss: 129000000.0
Cutoff: 0.4, Flight Loss: 123000000.0
Cutoff: 0.425, Flight Loss: 120400000.0
Cutoff: 0.45, Flight Loss: 119400000.0
Cutoff: 0.475, Flight Loss: 119100000.0
Cutoff: 0.5, Flight Loss: 118300000.0
Cutoff: 0.525, Flight Loss: 118400000.0
Cutoff: 0.55, Flight Loss: 118800000.0
Cutoff: 0.575, Flight Loss: 118100000.0
Cutoff: 0.6, Flight Loss: 118200000.0
Cutoff: 0.625, Flight Loss: 118200000.0
Cutoff: 0.65, Flight Loss: 118100000.0
Cutoff: 0.675, Flight Loss: 118300000.0
Cutoff: 0.7, Flight Loss: 118300000.0
Cutoff: 0.725, Flight Loss: 118400000.0
Cutoff: 0.75, Flight Loss: 118400000.0
```

The code above runs a loop for each cutoff value we would like to test. It takes each cutoff value and evaluates them using a model to determine the best cutoff value.

In [475…

```python
# Find the best cutoff based on accuracy
cutoff = None
min_loss = 1000000000000000000000000000000

for result in results:
    if result['loss'] < min_loss:
        min_loss = result['loss']
        cutoff = result['cutoff']

print(f"Best Cutoff Value: {cutoff}")
print(f"Minimum lost for Best Cutoff: ${min_loss:,.4f}")
LogRloss = min_loss
```

```
Best Cutoff Value: 0.575
Minimum lost for Best Cutoff: $118,100,000.0000
```

EVALUATE:

In [476…

```python
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
```

To evaluate our models, we will be using our loss function. Using our loss function allows us to capture the fact that the cost of a false negative, sending a plane that should have been delayed, is way higher than the cost of a false positive, delaying a flight that didn't need to be delayed.

Ridge Regression

The first penalized regression function we are going to run is ridge regression.

RECONCILE:

Since Ridge Regression is just a specialized form of Generalized Linear Regression, all of our assumptions are met.

FIT:

In [477...
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

In [478...
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import make_scorer

# Custom scorer wrapper
def flight_loss_scorer(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    fp = cm[0, 1]
    fn = cm[1, 0]
    return flight_loss(fp, fn)

# Tell sklearn this is a MINIMIZATION metric
flight_scorer = make_scorer(flight_loss_scorer, greater_is_better=False)
```

In [479...
```
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchC
V
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.utils import resample
from sklearn.linear_model import LogisticRegression

# Create a pipeline
ridge_pipe = Pipeline([
  ('feature_engineering', StandardScaler()),
  ('classification', LogisticRegression(penalty = 'l2'))
])
```

In [480...
```
# Create a grid using a log scale (inverse of regularization strength)
hyper_grid = {'classification__C': np.logspace(-3, 3, 30)}

# Use the grid to tune hyperparameters via cross-validation
kfold_cv = StratifiedKFold(n_splits = 5)
tune = GridSearchCV(
  ridge_pipe, hyper_grid, scoring = flight_scorer,
  cv = kfold_cv, n_jobs = 1, refit = True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())

y_prob = tune.best_estimator_.predict_proba(X_test)[:, 1]
y_pred = (y_prob >= cutoff).astype(int)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
fp = cm[0, 1]
fn = cm[1, 0]

Ridgeloss = flight_loss(fp, fn)

best_C = tune.best_params_['classification__C']
```

EVALUATE:

In [481…

```
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
print(f'Ridge Loss: ${Ridgeloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
Ridge Loss: $254,500,000.00
```

Ridge Regression did slightly worse in the recall function. We can expect this from the penalized regression models as they tend to favor the larger class, and we are predicting on the smaller class in our distribution.

LASSO Regression

We are now going to run LASSO as our next form of penalized regression.

RECONCILE:

LASSO regression also uses the base of generalized linear regression without any extra assumptions, so all of our assumption checking from generalized linear regression still apply.

FIT:

In [482…

```
# Create a lasso pipeline
lasso_pipe = Pipeline([
  ('feature_engineering', StandardScaler()),
  ('classification', LogisticRegression(penalty = 'l1', solver = 'liblinear'))
])
```

In [483…

```
# Create a grid using a log scale (inverse of regularization strength)
hyper_grid = {'classification__l1_ratio': [0.1, 0.25, 0.5, 0.75, 0.9]}
```

```
# Use the grid to tune hyperparameters via cross-validation
kfold_cv = StratifiedKFold(n_splits = 5)
tune = GridSearchCV(
  lasso_pipe, hyper_grid, scoring = flight_scorer,
  cv = kfold_cv, n_jobs = 1, refit = True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())

# Extract the best hyperparameter and CV
best_l1 = tune.best_params_['classification__l1_ratio']
y_pred = tune.best_estimator_.predict_proba(X_test)[:, 1]
y_pred = (y_pred >= cutoff).astype(int)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
fp = cm[0, 1]
fn = cm[1, 0]

lassoloss = flight_loss(fp, fn)
```

EVALUATE:

In [484…

```
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
print(f'Ridge Loss: ${Ridgeloss:,.2f}')
print(f'LASSO Recall: ${lassoloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
Ridge Loss: $254,500,000.00
LASSO Recall: $256,500,000.00
```

LASSO Regression did slightly worse than our previous penalized regression format.It struggled with the fact that penalized regression favors the biggest class in the distribution.

Elastic Net Regression

Our final penalized regression we are going to run is an elastic net regression.

RECONCILE:

Once again, Elastic Net is just a specialized form of Generalized Linear Regression, therefore all of our assumptions are still met.

FIT:

In [485…

```
# Create an elastic net pipeline
elastic_pipe = Pipeline([
  ('feature_engineering', StandardScaler()),
  ('classification', LogisticRegression(
    penalty = 'elasticnet',
    solver = 'saga', max_iter=25, random_state=42
  ))
])

# Create a grid that includes both hyperparameters
hyper_grid = {
  'classification__C': np.logspace(-3, 3, 10),
  'classification__l1_ratio': [0.1, 0.25, 0.5, 0.75, 0.9]
}

# Use the grid to tune both hyperparameters via cross-validation
tune = GridSearchCV(
  elastic_pipe, hyper_grid, scoring = 'recall',
  cv = kfold_cv, n_jobs = 1, refit = True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())

# Extract the best hyperparameters and CV recall
best_C = tune.best_params_['classification__C']
best_l1 = tune.best_params_['classification__l1_ratio']
y_pred = tune.best_estimator_.predict_proba(X_test)[:, 1]
y_pred = (y_pred >= cutoff).astype(int)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
fp = cm[0, 1]
fn = cm[1, 0]

Elasticloss = flight_loss(fp, fn)
```

EVALUATE:

In [486…

```
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
print(f'Ridge Loss: ${Ridgeloss:,.2f}')
print(f'LASSO Loss: ${lassoloss:,.2f}')
print(f'ElasticNet Loss: ${Elasticloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
Ridge Loss: $254,500,000.00
LASSO Loss: $256,500,000.00
ElasticNet Loss: $254,500,000.00
```

The Elastic Net model we ran is tied with our ridge model for least loss out of our penalized models. We have two more models to run before we can choose the best model.

Principal Component Regression

We have tested a bunch of penalized regression functions, but we should also run a principal component analysis and apply it to our logit model.

RECONCILE:

In [487…

```
from sklearn.decomposition import PCA

# Create a pipeline
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(random_state=42)),
    ('classify', LogisticRegression(fit_intercept=True,
        penalty=None, class_weight='balanced')),
])
# Create a hyperparameter grid for number of PCA components
hyper_grid = {'pca__n_components': np.arange(1, 8, 1)}
# Use the grid to tune hyperparameters via cross-validation
kfold_cv = StratifiedKFold(n_splits=5)
tune = GridSearchCV(
    pcr_pipe, hyper_grid, scoring= flight_scorer,
    cv=kfold_cv, n_jobs=1, refit=True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())
# Extract the best hyperparameter and CV accuracy
best_n_components = tune.best_params_['pca__n_components']
best_cv_score = tune.best_score_
best_cv_score = best_cv_score * -1


print(
    f'Best K: {best_n_components}',
    f'Best CV Recall: ${best_cv_score:,.2f}',
    sep='\n'
)
```

```
Best K: 3
Best CV Recall: $154,500,000.00
```

5 fold cross validation says the optimal amount of principal componenents is 3. However, this doesn't account for uncertaintity, so we will graph the amount and try to find a lower amount of columns using one standard error to find other viable options.

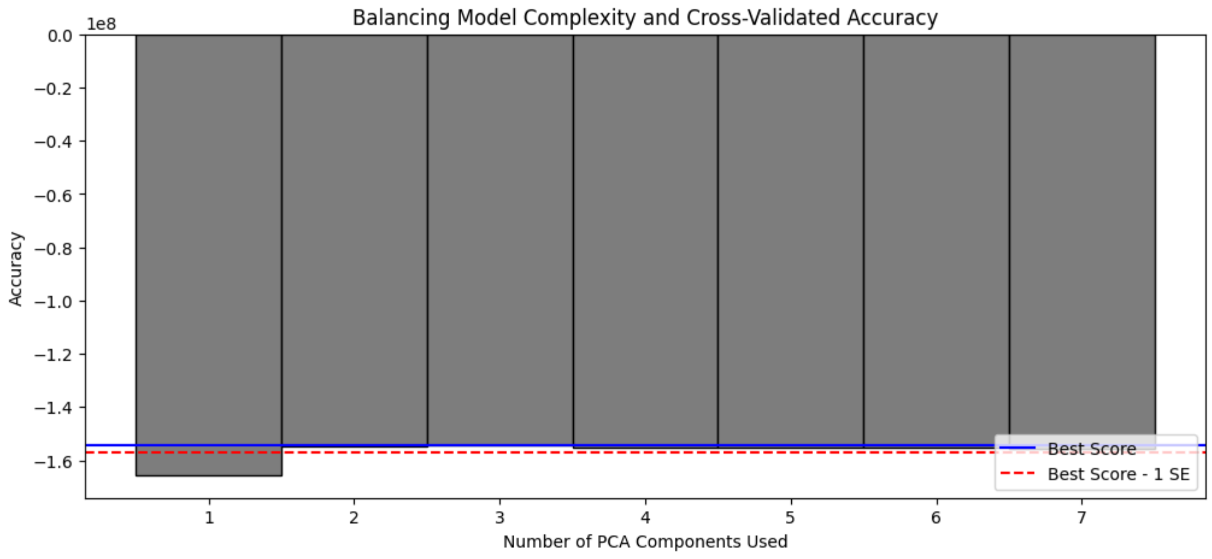
In [488…

```
def lower_bound(cv_results):
    best_score_idx = np.argmax(cv_results['mean_test_score'])
    return (
        cv_results['mean_test_score'][best_score_idx]
        - (cv_results['std_test_score'][best_score_idx] / np.sqrt(5)) # 5-fold CV
    )
def best_low_complexity(cv_results):
    threshold = lower_bound(cv_results)
    candidate_idx = np.flatnonzero(cv_results['mean_test_score'] >= threshold)
    best_idx = candidate_idx[
        cv_results['param_reduce_dim__n_components'][candidate_idx].argmin()
    ]
    return best_idx

# Extracted cross-validation results
```

```python
n_components = tune.cv_results_['param_pca__n_components']
mean_test_score = tune.cv_results_['mean_test_score']

# Visualize cross-validated results
plt.figure(figsize = (12, 5))
plt.bar(n_components,
        mean_test_score,
        width = 1,
        color = 'grey',
        edgecolor = 'black'
)

# Add lines for best score and one-standard-error rule
lower = lower_bound(tune.cv_results_)
plt.axhline(np.max(mean_test_score),
            linestyle = '-',
            color = 'blue',
            label = 'Best Score'
)
plt.axhline(lower,
            linestyle = '--',
            color = 'red',
            label = 'Best Score - 1 SE'
)
# Add title, labels, and legend
plt.title('Balancing Model Complexity and Cross-Validated Accuracy')
plt.xlabel('Number of PCA Components Used')
plt.ylabel('Accuracy')
plt.xticks(n_components.tolist())
plt.legend(loc = 'lower right')
```

Out[488…

```
<matplotlib.legend.Legend at 0x2079a12e910>
```



We can see that two principal components is a possible value within the 1 error rule, so we will run the model using 2 principal components.

FIT:

In [489…

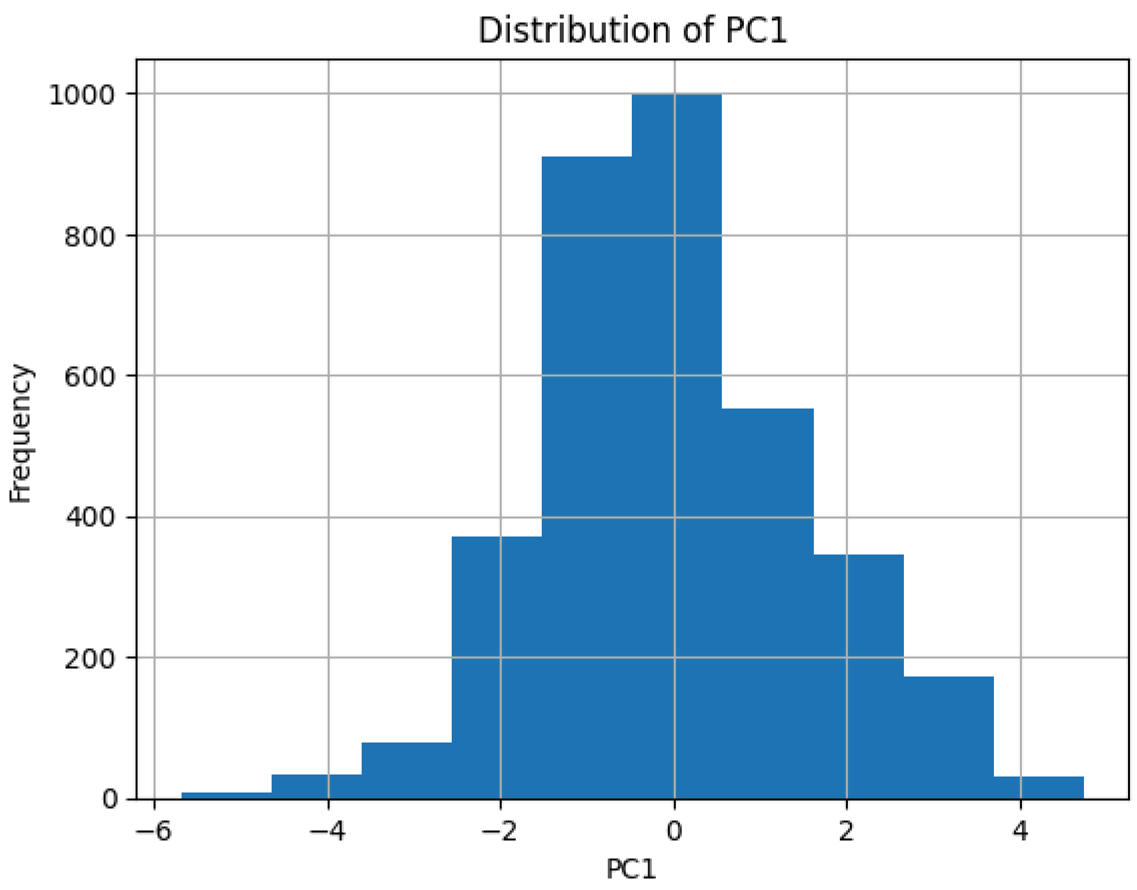
```
# Update the pipeline with n_components = 2
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components = 2, random_state = 42)),
    ('classify', LogisticRegression(fit_intercept = True, penalty = None)),
])
# Specify a new estimator that allows for cutoff tuning
from sklearn.base import BaseEstimator, ClassifierMixin

class CutoffClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, pipeline, cutoff = 0.5):
        self.pipeline = pipeline
        self.cutoff = cutoff

    def fit(self, X_train, y_train):
        self.pipeline.fit(X_train, y_train)
        return self

    def predict_proba(self, X_train):
        return self.pipeline.predict_proba(X_train)

    def predict(self, X_train):
        y_pred_proba = self.predict_proba(X_train)[:, 1]
        return (y_pred_proba >= self.cutoff).astype(int)
```

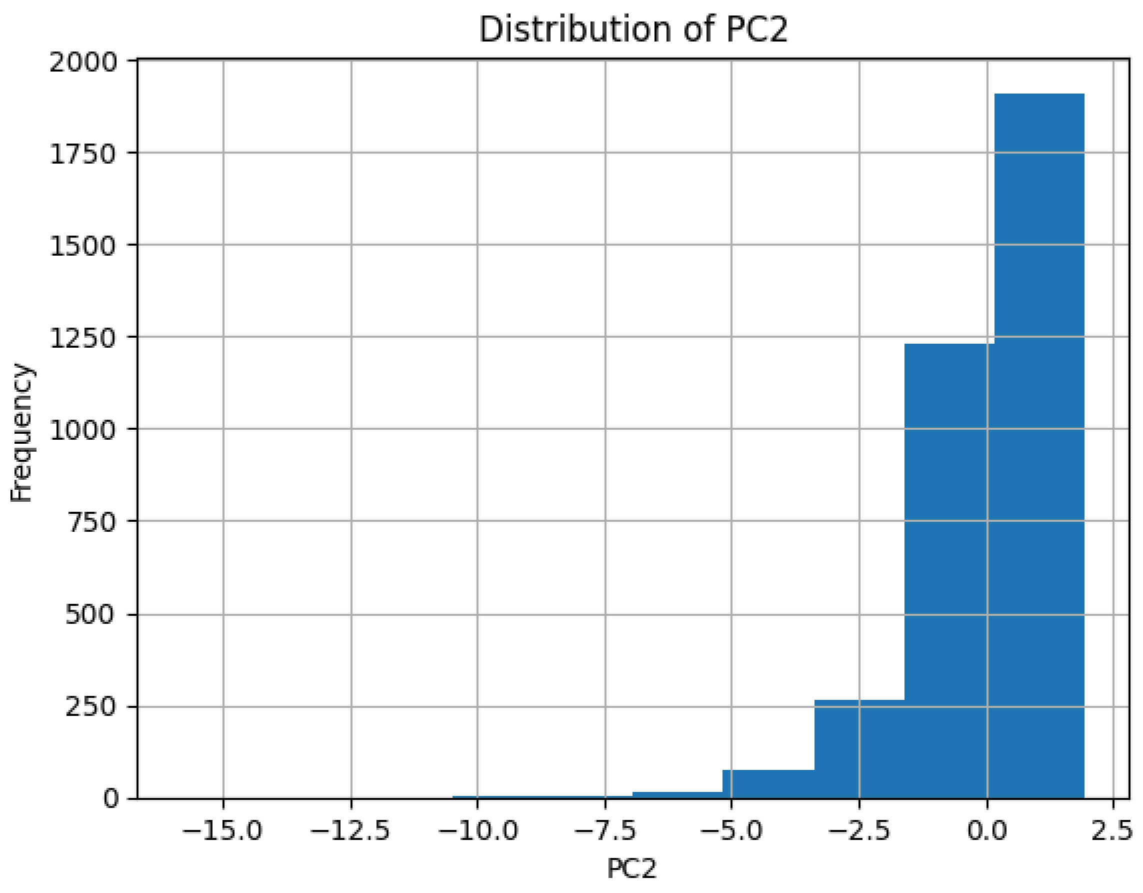
We are going to rerun our model diagnostics to see if our principal components still fit our model assumptions for logistic regression.

In [490…

```
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# ----------------------------
# 1. Fit the PCR pipeline
# ----------------------------
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=2, random_state=42)),
    ('classify', LogisticRegression(fit_intercept=True, penalty=None)),
])

pcr_pipe.fit(X_train, y_train)

# ----------------------------
# 2. Get principal component scores
# ----------------------------
X_scaled = pcr_pipe.named_steps['scaler'].transform(X_train)
X_pca = pcr_pipe.named_steps['pca'].transform(X_scaled)
```

```
pc_df = pd.DataFrame(X_pca, columns=["PC1", "PC2"])
pc_df["Delayed"] = y_train.values

# --- PC1 Distribution ---
plt.figure()
pc_df["PC1"].hist()
plt.title("Distribution of PC1")
plt.xlabel("PC1")
plt.ylabel("Frequency")
plt.show()

# --- PC2 Distribution ---
plt.figure()
pc_df["PC2"].hist()
plt.title("Distribution of PC2")
plt.xlabel("PC2")
plt.ylabel("Frequency")
plt.show()
```





In [491…

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# -----------------------------
# 1. Fit the PCR pipeline
# -----------------------------
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=2, random_state=42)),
    ('classify', LogisticRegression(fit_intercept=True, penalty=None)),
])

pcr_pipe.fit(X_train, y_train)

# -----------------------------
# 2. Get principal component scores
# -----------------------------
X_scaled = pcr_pipe.named_steps['scaler'].transform(X_train)
X_pca = pcr_pipe.named_steps['pca'].transform(X_scaled)

pc_df = pd.DataFrame(X_pca, columns=["PC1", "PC2"])
pc_df["Delayed"] = y_train.values
```

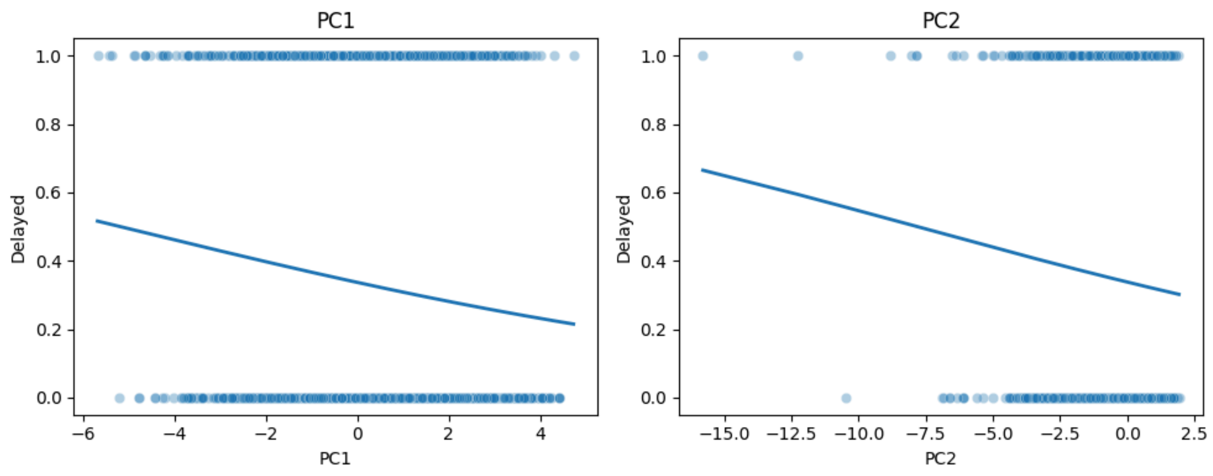
```
# ----------------------------
# 3. Fit statsmodels logit using PCs
# ----------------------------
model_formula = "Delayed ~ PC1 + PC2"
logit_model = smf.glm(
    formula=model_formula,
    data=pc_df,
    family=sm.families.Binomial()
).fit()

# ----------------------------
# 4. Influence & residuals
# ----------------------------
influence = logit_model.get_influence()
summary = influence.summary_frame()

# Standardized residuals
standard_resid = summary['standard_resid']

# Residuals greater than 2
resid_gt_2 = standard_resid[np.abs(standard_resid) > 2]

print("\nStandardized Residuals | > 2 |")
print(resid_gt_2)
```

```
Standardized Residuals | > 2 |
Series([], Name: standard_resid, dtype: float64)
```

In [492…

```
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# ---- Fit PCR pipeline ----
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=2, random_state=42)),
    ('classify', LogisticRegression(fit_intercept=True, penalty=None)),
])

pcr_pipe.fit(X_train, y_train)

# Transform X into PC space
X_scaled = pcr_pipe.named_steps['scaler'].transform(X_train)
X_pca = pcr_pipe.named_steps['pca'].transform(X_scaled)

# Put in dataframe for easier plotting
pc_df = pd.DataFrame(X_pca, columns=["PC1", "PC2"])
pc_df["Delayed"] = y_train.values

cols = 2
```

```
rows = 1
plt.figure(figsize=(10,4))

for i, col in enumerate(["PC1", "PC2"]):
    plt.subplot(rows, cols, i+1)

    x = pc_df[col]
    y = pc_df["Delayed"]

    # Scatter
    sns.scatterplot(x=x, y=y, alpha=0.35)

    # Remove missing
    mask = ~(x.isna() | y.isna())
    x_clean = x[mask]
    y_clean = y[mask]

    # Fit logistic model
    X_sm = sm.add_constant(x_clean)
    model = sm.Logit(y_clean, X_sm).fit(disp=0)

    # Sorted line
    sort_idx = np.argsort(x_clean)
    x_sorted = x_clean.iloc[sort_idx]
    pred_sorted = model.predict(X_sm.iloc[sort_idx])

    sns.lineplot(x=x_sorted, y=pred_sorted, linewidth=2)

    plt.title(col)
    plt.ylim(-0.05, 1.05)

plt.tight_layout()
plt.show()
```



The data remains valid and independent. Since principal components are orthogonal in hyperspace, there are no issues with multicollinearity. Our representativeness check shows no values with residuals greater than 2, meaning that our data is representative. We also see that the relationship of our principal components is linear and negative, so the assumption of linearity still holds. Our first principal component is normally distributed, and while our second principal component has some skew, we did not choose to transform it. All of our model assumptions are checked, so we will move on with the model.

In [493…

```
# Wrap the new estimator around the pipeline
cutoff_pipe = CutoffClassifier(pipeline = pcr_pipe)
# Create a hyperparameter grid for classification cutoff
hyper_grid = {'cutoff': np.linspace(0.3, 0.9, 81)}
# Use the grid to tune hyperparameters via cross-validation
kfold_cv = StratifiedKFold(n_splits = 5)
tune = GridSearchCV(
    cutoff_pipe, hyper_grid, scoring = flight_scorer,
    cv = kfold_cv, n_jobs = 1, refit = True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())
# Extract the best hyperparameter and CV
PCRcutoff = tune.best_params_['cutoff']
best_cv_loss = tune.best_score_
PCRloss = best_cv_loss * -1

print(
    f'Best Cutoff: {PCRcutoff:.3f}',
    f'Best CV Flight Loss: {PCRloss:.0f}',
    sep='\n'
)
```

```
Best Cutoff: 0.480
Best CV Flight Loss: 117500000
```

EVALUATE:

In [494…

```
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
print(f'Ridge Loss: ${Ridgeloss:,.2f}')
print(f'LASSO Loss: ${lassoloss:,.2f}')
print(f'ElasticNet Loss: ${Elasticloss:,.2f}')
print(f'PCR Loss: ${PCRloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
Ridge Loss: $254,500,000.00
LASSO Loss: $256,500,000.00
ElasticNet Loss: $254,500,000.00
PCR Loss: $117,500,000.00
```

Our principal component regression outperformed our other models and did slightly better than our Logistic Regression model, so we chose it as our final model.

Interactions

Our current data set doesn't account for any interactions among the independent variables, so with this next model, we are going to implement an interaction.

RECONCILE:

We are going to fit our best model, PCR, with an interaction included to see if that improves our model.

FIT:

In [495…

```
pcr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components = 2, random_state = 42)),
    ('classify', LogisticRegression(fit_intercept = True, penalty = None)),
])

# Wrap the new estimator around the pipeline
cutoff_pipe = CutoffClassifier(pipeline = pcr_pipe)
# Create a hyperparameter grid for classification cutoff
hyper_grid = {'cutoff': np.linspace(0.3, 0.9, 81)}
# Use the grid to tune hyperparameters via cross-validation
kfold_cv = StratifiedKFold(n_splits = 5)
tune = GridSearchCV(
    cutoff_pipe, hyper_grid, scoring = flight_scorer,
    cv = kfold_cv, n_jobs = 1, refit = True, verbose=0
)
tune.fit(X_train, y_train.to_numpy().ravel())
# Extract the best hyperparameter and CV
PCRIcutoff = tune.best_params_['cutoff']
best_cv_loss = tune.best_score_
PCRIloss = best_cv_loss * -1

print(
    f'Best Cutoff: {PCRIcutoff:.3f}',
    f'Best CV Flight Loss: {PCRIloss:.0f}',
    sep='\n'
)
```

```
Best Cutoff: 0.480
Best CV Flight Loss: 117500000
```

EVALUATE:

In [496…

```
print(f'Logistic Regression Loss: ${LogRloss:,.2f}')
print(f'Ridge Loss: ${Ridgeloss:,.2f}')
print(f'LASSO Loss: ${lassoloss:,.2f}')
print(f'ElasticNet Loss: ${Elasticloss:,.2f}')
print(f'PCR Loss: ${PCRloss:,.2f}')
print(f'PCR with Interaction Loss: ${PCRIloss:,.2f}')
```

```
Logistic Regression Loss: $118,100,000.00
Ridge Loss: $254,500,000.00
LASSO Loss: $256,500,000.00
ElasticNet Loss: $254,500,000.00
PCR Loss: $117,500,000.00
PCR with Interaction Loss: $117,500,000.00
```

It seems like our best model was our original PCR model. We will run that on our test data.

PREDICT

FIT:

In [497…

```python
fixed_cutoff = 0.48

# Create a model with the fixed cutoff
test_model = CutoffClassifier(
    pipeline=pcr_pipe,
    cutoff=fixed_cutoff
)

# Fit model on training data
test_model.fit(X_train, y_train.to_numpy().ravel())

# Predict on test set
y_pred_test = test_model.predict(X_test)
```

EVALUATE:

In [498…

```python
# Compute flight loss
cm = confusion_matrix(y_test, y_pred_test)
fp = cm[0, 1]
fn = cm[1, 0]

test_loss = flight_loss(fp, fn)

print(f"\nTest Flight Loss using cutoff {fixed_cutoff}: ${test_loss:,.2f}")
```

```
Test Flight Loss using cutoff 0.48: $252,500,000.00
```

In [499…

```python
# Running code to apply our loss function on a baseline assumption,
# which would be assuming every flight as a non-delayed flight.

baseline_loss_value = flight_loss(
  false_positives = 0, #we have no false positives,
    # as we are assuming every flight to not have a delay
  false_negatives = (len(df_sampled) * .33) #33% of our data is delays,
    # meaning 33% will be false negatives
)
```

```
print(f"Baseline Flight Loss: ${baseline_loss_value:,.2f}")
```

```
Baseline Flight Loss: $825,000,000.00
```

In [500…

```
Saving = baseline_loss_value - test_loss
print(f"Saving: ${Saving:,.2f}")
```

```
Saving: $572,500,000.00
```

Using our model we could save Salt Lake City Airport **$572,500,000**.

Now we want to output the interval estimates for our principal components, but first, we need to name them.

In [501…

```
pca = test_model.pipeline.named_steps['pca']
pca_loadings = (
    pl.DataFrame(pca.components_).transpose()
    .rename({'column_0': 'PC1', 'column_1': 'PC2'})
    .with_columns(pl.Series('predictors', X_train.columns.tolist()))
)
```

In [502…

```
# Sort the loadings for PC1
pca_loadings.select(['predictors', 'PC1']).sort('PC1', descending =
True).head(5)
```

Out[502…

shape: (5, 2)

predictors	PC1
str	f64
"Sea_Level_Pressure"	0.612449
"Altimeter_Pressure"	0.567805
"Visibility"	0.019215
"Ice_Accretion_3hr"	-0.037231
"Precipitation"	-0.08974

Based on these loadings, we want to call PC1 "Atmospheric Pressure"

In [503…

```
# Sort the loadings for PC2
pca_loadings.select(['predictors', 'PC2']).sort('PC2', descending =
True).head(5)
```

Out[503... shape: (5, 2)

predictors	PC2
str	f64
"Visibility"	0.658023
"Temperature"	0.472904
"Altimeter_Pressure"	0.165158
"Wind_Speed"	-0.037792
"Sea_Level_Pressure"	-0.056655

Based on these loadings, we want to call PC2 “Visibility & Temperature”

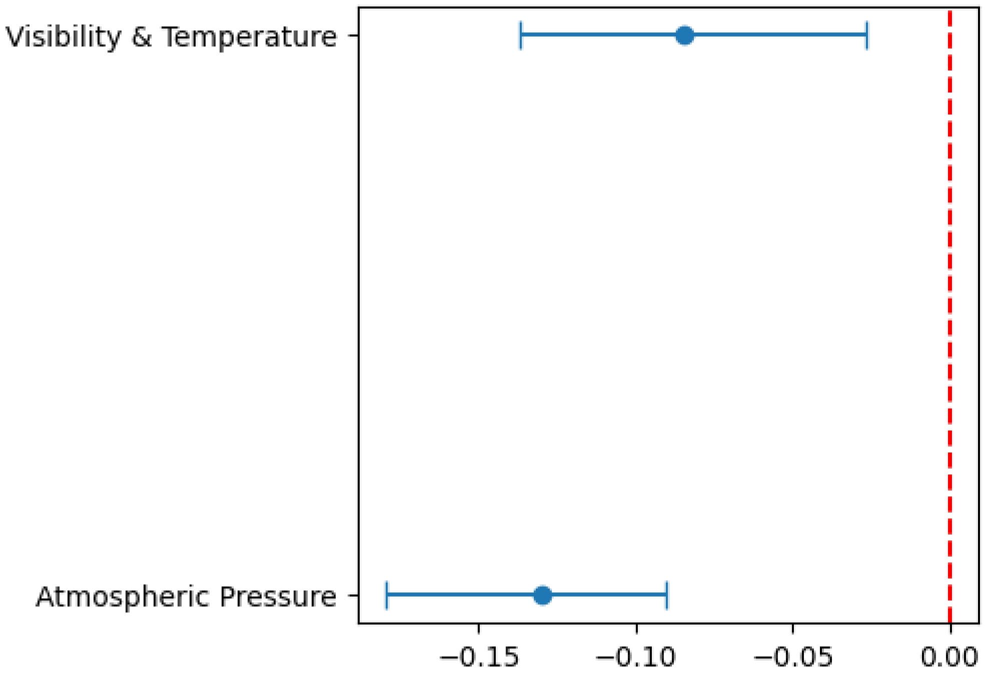
In [504...
```
pca_loadings = (
    pl.DataFrame(pca.components_).transpose()
    .rename({'column_0': 'Atmospheric Pressure',
        'column_1': 'Visibility & Temperature'})
    .with_columns(pl.Series('predictors', X_train.columns.tolist()))
)
```

In [505...
```
# Extract point estimates from refit best_estimator
pcr_final = test_model.pipeline.named_steps['classify']
pcr_intercept = pcr_final.intercept_.ravel()
pcr_slopes = pcr_final.coef_.ravel()
pcr_point_est = np.concatenate([pcr_intercept, pcr_slopes])

# Bootstrap confidence intervals
n_samples = 100
pcr_boot_est = np.empty((n_samples, len(pcr_point_est)))
for b in range(n_samples):
    # Resample data with replacement
    X_b, y_b = resample(X_train, y_train, replace = True, random_state = 42 + b)
    # Extract and save point estimates for PCR
    boot_pcr_pipe = test_model.pipeline
    # Use the pipeline from the fitted test_model
    boot_pcr_pipe.fit(X_b, y_b.to_numpy().ravel())
    pcr_b = boot_pcr_pipe.named_steps['classify']
    pcr_intercept_b = pcr_b.intercept_.ravel()
    pcr_slopes_b = pcr_b.coef_.ravel()
    pcr_point_est_b = np.concatenate([pcr_intercept_b, pcr_slopes_b])
    pcr_boot_est[b, :] = pcr_point_est_b
```

In [506...
```
pcr_int_est = pl.DataFrame({
    'predictors': ['Intercept'] + pca_loadings.columns[:2],
    'point_est': pcr_point_est,
    'ci_lower': np.percentile(pcr_boot_est, 2.5, axis=0),
    'ci_upper': np.percentile(pcr_boot_est, 97.5, axis=0)
}).filter(pl.col('predictors') != 'Intercept')

# Plot the confidence intervals
plt.figure(figsize=(4, 4))
plt.errorbar(
    pcr_int_est['point_est'],
    pcr_int_est['predictors'],
    xerr=[
        pcr_int_est['point_est'] - pcr_int_est['ci_lower'],
        pcr_int_est['ci_upper'] - pcr_int_est['point_est']
    ],
    fmt='o',
    capsize=5,
    label='Estimates'
)
plt.axvline(0, color='red', linestyle='--', label='y=0')
```

Out[506…

```
<matplotlib.lines.Line2D at 0x207883b3b10>
```



In the above chart, both of our principal components are negatively correlated with delaying a flight.

In [507…

```
print(pcr_int_est)
```

```
shape: (2, 4)
┌──────────────────────────┬───────────┬───────────┬───────────┐
│ predictors               ┆ point_est ┆ ci_lower  ┆ ci_upper  │
│ ---                      ┆ ---       ┆ ---       ┆ ---       │
│ str                      ┆ f64       ┆ f64       ┆ f64       │
╞══════════════════════════╪═══════════╪═══════════╪═══════════╡
│ Atmospheric Pressure     ┆ -0.129758 ┆ -0.179708 ┆ -0.090525 │
│ Visibility & Temperature ┆ -0.08437  ┆ -0.136874 ┆ -0.026284 │
└──────────────────────────┴───────────┴───────────┴───────────┘
```

Interpretation

We are 95% confident that the true effect of a one unit increase in Visibility & Temperature on the log-odds of the outcome lies within -0.179708 and -0.090525. This component has statistical significance.

We are 95% confident that the true effect of a one unit increase in Visibility & Temperature on the log-odds of the outcome lies within -0.136874 and -0.026284. This component has statistical significance.