Phase 6: Data Optimization

Team name: **THE TRAILBLAZERS**
Team members:

1) Vyshnavi Basude– vbasu1@unh.newhaven.edu
2) Manasa Sukavasi(msuka1@newhaven.edu)
3) Pooja Donuru(pdonu1@unh.newhaven.edu)

GIT Repository link: https://github.com/VyshnaviBasude/Team-TrailBlazers

Link for source code:

https://github.com/VyshnaviBasude/Team-TrailBlazers/blob/main/Data%20Modeling_Techniques.ipynb

1. Introduction

Machine learning algorithms power Uber's dynamic pricing strategy by processing historical ride data, weather forecasts, event calendars, and traffic conditions. These algorithms predict when and where demand for rides will increase, allowing Uber to allocate more drivers to high-demand areas and optimize pricing accordingly. Surge pricing incentivizes drivers to meet surging demand while ensuring efficient service for passengers. This data-driven approach maximizes driver earnings during peak times and enhances overall user satisfaction by reducing wait times and providing reliable transportation options.

2. Dataset:

The dataset is called the " Uber and Lyft Dataset Boston, MA" and it contains information about the ride information between Uber and Lyft . Here are some details about the dataset:

a. The selected dataset contains extensive details regarding Uber rides taken over a two month period in Boston, Massachusetts.
b. It provides not only ride details but also contextual features such as the weather conditions, time specifics, and environmental settings during each trip. Accessibility: The dataset is publicly available on Kaggle, a platform for predictive modeling and analytics competitions. Users can freely download this dataset after creating an account on Kaggle.
c. The data in this dataset has been collected using APIs provided by both Uber and Lyft, amalgamated with weather data APIs to provide context regarding environmental conditions during each ride.
d. The dataset is likely to have categorical data representing the type of ride (e.g., UberX, UberXL), although this is inferred and not explicitly mentioned in the provided data.

Exploring Uber and Lyft , ride sharing details, pricing , temperature , hours to Drive Sales. Identification of Deviations in sales and supply chain based on Customer reviews andsentiment analysis.

The columns which we have in our dataset are:
Temporal Data: hour, day, month, sunriseTime, sunsetTime, windGustTime, temperatureHighTime, temperatureLowTime, apparentTemperatureHighTime, apparentTemperatureLowTime, uvIndexTime, temperatureMinTime, temperatureMaxTime, apparentTemperatureMinTime, apparentTemperatureMaxTime,
Numerical Data: price, distance, temperature, apparentTemperature, precipIntensity, precipProbability, humidity, windSpeed, windGust, visibility, temperatureHigh, temperatureLow, apparentTemperatureHigh, apparentTemperatureLow, dewPoint, pressure, windBearing, cloudCover, uvIndex, ozone, moonPhase, precipIntensityMax.
Categorical Data: The dataset is likely to have categorical data representing the type of ride (e.g., UberX, UberXL), although this is inferred and not explicitly mentioned in the provided details.

3. Data Mining Techniques:

The Following are the Data Mining techniques to forecast future sales prediction:

- Logistic Regression
- Naïve Bayes
- Decision Tree Regression
- KNN

Logistic Regression:
Type: Classification
Logistic Regression is a statistical method used for predicting a binary outcome (1 / 0, True / False, Yes / No) based on one or more predictor variables. In the context of sales prediction, it can be used for classifying whether a customer is likely to make a purchase (yes or no) based on features like customer demographics, purchase history, or other relevant factors. It's not typically used for predicting actual sales values but rather for binary classification tasks.

Naive Bayes:
Type: Classification
Naïve Bayes is a probabilistic classification algorithm based on Bayes' theorem. It is particularly useful for text classification problems but can be adapted for sales prediction. It estimates the probability of a given event based on prior knowledge of conditions that might be related to that event. In the context of sales, it could be used to predict whether a customer will buy a product or not based on various features such as product attributes, customer behavior, and more.

Decision Tree Regression:
Type: Regression
Decision Tree Regression is a tree-based machine learning algorithm used for both classification and regression tasks. In the context of sales prediction, it is used for predicting the numerical value of sales (e.g., revenue) based on a set of input features (e.g., time, advertising spending, product features). Decision trees recursively split the data into subsets and make predictions at the leaves of the tree. It's a versatile algorithm for forecasting sales when the relationships between features and sales are non-linear or complex.

K-Nearest Neighbors (KNN):
Type: Both Classification and Regression
K-Nearest Neighbors is a versatile algorithm that can be used for both classification and regression tasks. For sales prediction, it can be applied in regression mode to predict future sales values based on the values of K nearest neighbors in the feature space. It works on the principle that similar data points tend to have similar outcomes. In the context of sales, it can be used to find similar sales scenarios and predict future sales based on those similarities.

4. Parameters and Hyperparameters

Parameters:

Data Mining techniques	Parameters	Data Model
Logistic Regression	Weights/Coefficients	Classification
Naïve Bayes	Accuracy	Classification
Decision Trees	Tree Structure Feature Weights	Regression
KNN	Accuracy	Both Classification and Regression

Hyper parameters:

Data Mining techniques	Hyper parameters	Values Used
Logistic Regression	solver,class_weight, max_iter=100000	"liblinear",'balanced',100000
Naïve Bayes	Default	Default
Decision Trees	max_depth min_samples_split	10,4
KNN	N_neighbours	5

5 .Optimization

To observe how our model is reacting to different kinds of optimization technique we have tried to do the Boosting ,GridSearch, Randomized Search

Boosting:

Boosting is an ensemble learning technique where multiple weak learners are combined to form a strong learner. It is done sequentially, with each model correcting the errors of its predecessor.

Popular boosting algorithms include AdaBoost, Gradient Boosting (e.g., XGBoost, LightGBM), and CatBoost. These algorithms are widely used for classification and regression tasks.

For our model We have used

XG Boost:

```
pip install xgboost
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.10/dist-packages (2.0.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.23.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.11.3)
```

```
# Drop non-numeric columns
non_numeric_columns = ['id', 'datetime', 'timezone', 'source', 'product_id', 'name', 'short_summary', 'long_summary', 'icon']
df_numeric = df.drop(non_numeric_columns, axis=1)
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report


# Step 1: Check Label Values
print(df['destination'].unique())

# Step 2: Label Encoding (Optional)
label_encoder = LabelEncoder()
df['destination'] = label_encoder.fit_transform(df['destination'])

# Check unique values after encoding
print(df['destination'].unique())

# Encode the target variable
df['cab_type'] = label_encoder.fit_transform(df['cab_type'])

# Drop non-numeric columns
non_numeric_columns = ['id', 'datetime', 'timezone', 'source', 'product_id', 'name', 'short_summary', 'long_summary', 'icon']
df_numeric = df.drop(non_numeric_columns, axis=1)

# Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(df_numeric.drop('cab_type', axis=1), df_numeric['cab_type'], test_size=0.2, random_state=42)

# Step 5: Convert Data to DMatrix
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Step 6: XGBoost Parameters
param_grid = {'objective': 'binary:logistic', 'eval_metric': 'logloss'}
# Step 7: Train XGBoost Model
num_rounds = 100
model = xgb.train(param_grid, dtrain, num_rounds)

# Step 8: Make Predictions
predictions = model.predict(dtest)
predictions_binary = [1 if p > 0.5 else 0 for p in predictions]
# Step 9: Evaluate Accuracy
accuracy = accuracy_score(y_test, predictions_binary)
print("Accuracy: {:.2f}%".format(accuracy * 100))
# Step 10: Classification Report
class_report = classification_report(y_test, predictions_binary)
print("Classification Report:\n", class_report)
```

Accuracy:

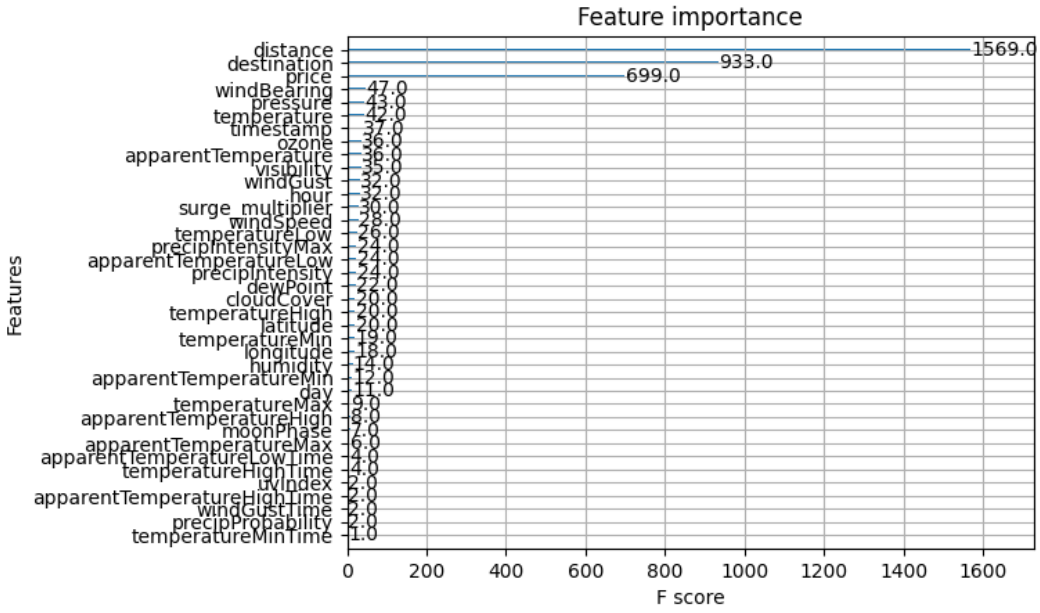
```
[ 7  8 11  5  9  3 10  1  0  6  4  2]
[ 7  8 11  5  9  3 10  1  0  6  4  2]
Accuracy: 95.70%
Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.97      0.96     61339
           1       0.97      0.94      0.96     66257

    accuracy                           0.96    127596
   macro avg       0.96      0.96      0.96    127596
weighted avg       0.96      0.96      0.96    127596
```

Using the XG Boost algorithm, we were able to optimize the model to 95% . The critical factors influencing the decision-making process of the model include destination,price, and distance. Through comprehensive analysis, it is evident that these features play a pivotal role in shaping the model's decisions.

Plotting the graph:

```
xgb.plot_importance(model)
```

```
<Axes: title={'center': 'Feature importance'}, xlabel='F score', ylabel='Features'>
```



ADA Boost:

```
[49] from sklearn.ensemble import AdaBoostClassifier
     from sklearn.tree import DecisionTreeClassifier
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import accuracy_score
```

```
[63] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

     base_classifier = DecisionTreeClassifier(max_depth=5)

     # AdaBoost model using the base Learner
     adaboost_model = AdaBoostClassifier(base_classifier, n_estimators=50, random_state=42)
     adaboost_model.fit(X_train, y_train)
```

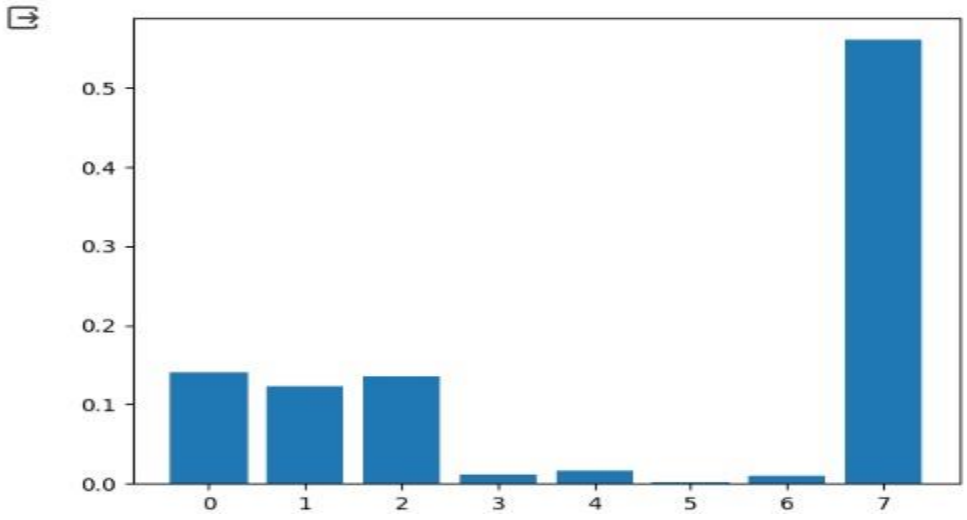
AdaBoostClassifier
estimator: DecisionTreeClassifier
DecisionTreeClassifier

```
predictions = adaboost_model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"accuracy: {accuracy}" )
```

```
accuracy: 0.9759240101570582
```

We attained a commendable accuracy of 97.59 using ADA Boost. The graph below depicts the importance of numerous elements in the decision-making process, with the X-axis showing the index of features and their respective importance. This visual representation provides useful insights into the algorithm's feature prioritizing.

```
if isinstance(base_classifier, DecisionTreeClassifier):
  feature_importance = adaboost_model.feature_importances_
  plt.bar(range(len(feature_importance)),feature_importance)
  plt.show()
```



Grid Search:

Grid search is a hyperparameter tuning technique that exhaustively searches a predefined set of hyperparameter values to find the best combination for a given model.It helps to find the optimal hyperparameters for a machine learning model by evaluating performance on a grid of parameter values.

While we were modelling our dataset using Decision Tree, as shown in below figure we got 52% accuracy,

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


feature_columns = ['hour', 'day', 'month', 'temperatureMin', 'temperatureMax']
target_variable = 'cab_type'

# Create feature matrix (X) and target variable (y)
X = df[feature_columns]
y = df[target_variable]
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize a Decision Tree classifier with specified hyperparameters
# max_depth controls the maximum depth of the tree
# min_samples_split is the minimum number of samples required to split an internal node
# You can adjust these values based on your dataset and desired complexity of the tree
decision_tree = DecisionTreeClassifier(max_depth=10, min_samples_split=4)

# Train the Decision Tree classifier
decision_tree.fit(X_train, y_train)

# Predict the labels on the test set
y_pred = decision_tree.predict(X_test)

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy
print("Accuracy:", accuracy)
print('\nClassification Report:\n', classification_report(y_test,y_pred))
```

```
Accuracy: 0.5207530016614941

Classification Report:
              precision    recall  f1-score   support

        Lyft       0.51      0.08      0.13     61339
        Uber       0.52      0.93      0.67     66257

    accuracy                           0.52    127596
   macro avg       0.52      0.50      0.40    127596
weighted avg       0.52      0.52      0.41    127596
```

Optimizing a Decision Tree involves tuning its hyperparameters and addressing potential issues with overfitting or underfitting. We have applied some common optimization techniques for

Decision Trees:

Tune Hyperparameters

Max Depth (max_depth): Controls the maximum depth of the tree. A deeper tree can capture more complex relationships in the data, but it may lead to overfitting.

Min Samples Split (min_samples_split): Minimum number of samples required to split an internal node. Increasing this value can prevent the tree from splitting too early, which may help control overfitting.

Min Samples Leaf (min_samples_leaf): Minimum number of samples required to be in a leaf node. Increasing this value can smooth the model and prevent overfitting.

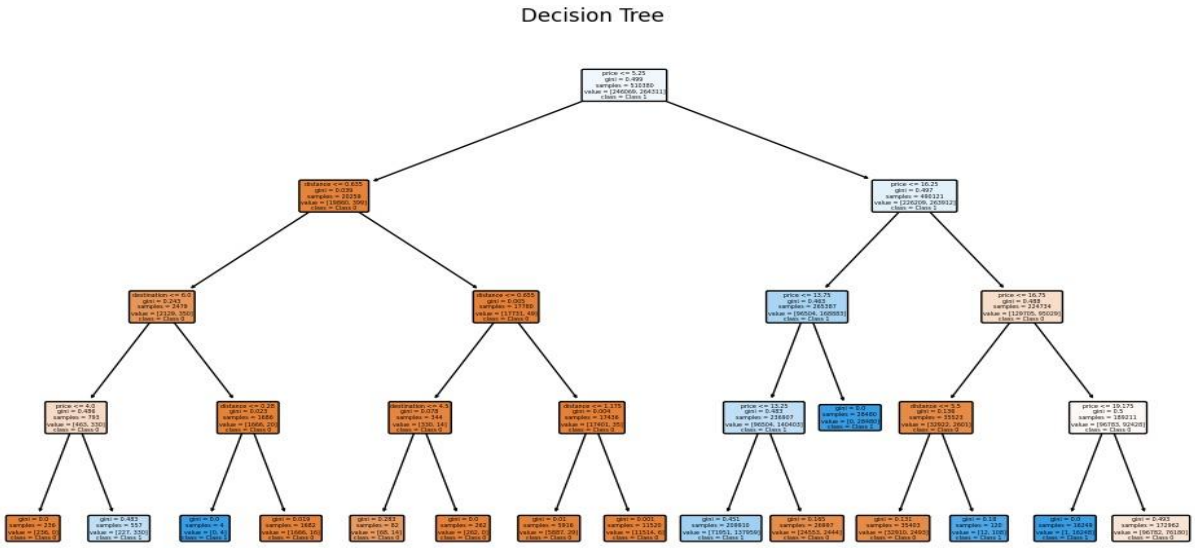
Max Features (max_features): Maximum number of features considered for splitting a node. This parameter can be adjusted to control the diversity of each split.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score

# Assuming your dataset is in a DataFrame called 'df'
feature_columns = ['source', 'price', 'destination', 'temperatureMin', 'temperatureMax']
target_variable = 'cab_type'

# Create feature matrix (X) and target variable (y)
X = df[feature_columns]
y = df[target_variable]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize Decision Tree classifier
decision_tree = DecisionTreeClassifier()

# Define hyperparameters for grid search
param_grid = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(decision_tree, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best model from the search
best_model = grid_search.best_estimator_

# Train the best model on the full training set
best_model.fit(X_train, y_train)
# Predict on the test set
y_pred = best_model.predict(X_test)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
# Print best hyperparameters, accuracy, and classification report
print('Best Hyperparameters:', grid_search.best_params_)
print(f'Accuracy: {accuracy}')
```

```
Best Hyperparameters: {'max_depth': None, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_s
Accuracy: 0.8386940029468009
```

By using Grid search , we have optimized it from 52% to 83% which is really good in terms of optimizting our accuracy.

Graph :



Randomized Search:

Randomized search is a hyperparameter tuning technique that randomly samples a fixed number of hyperparameter combinations from a predefined search space. It's more computationally efficient than grid search and can be particularly useful when the search space is large.

We tried to implement the randomized search optimization technique on KNN Model which we used in last phase, where we got 50% accuracy .

```
[48] from sklearn.neighbors import KNeighborsClassifier
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import accuracy_score

     # Assuming X and y are your features and labels respectively
     # You may need to preprocess your data first if it's not ready for modeling
     # Assuming X and y are your features and labels respectively
     # You may need to preprocess your data first if it's not ready for modeling
     feature_columns = ['source', 'price', 'destination', 'temperatureMin', 'temperatureMax', "latitude", "longitude", "distance"]
     target_variable = 'cab_type'

     # Create feature matrix (X) and target variable (y)
     X = df[feature_columns]
     y = df[target_variable]
     # Split the data into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

     # Initialize a KNN classifier with specified hyperparameters
     # n_neighbors is the number of neighbors to consider
     # You can adjust this value based on your dataset
     knn = KNeighborsClassifier(n_neighbors=5)

     # Train the KNN classifier
     knn.fit(X_train, y_train)

     # Predict the labels on the test set
     y_pred = knn.predict(X_test)

     # Calculate the accuracy
     accuracy = accuracy_score(y_test, y_pred)

     # Print the accuracy
     print("Accuracy:", accuracy)
     print('\nClassification Report:\n', classification_report(y_test,y_pred))
```

```
Accuracy: 0.5007758863914229

Classification Report:
              precision    recall  f1-score   support

        Lyft       0.48      0.49      0.48     61339
        Uber       0.52      0.51      0.52     66257

    accuracy                           0.50    127596
   macro avg       0.50      0.50      0.50    127596
weighted avg       0.50      0.50      0.50    127596
```

Now, To optimize we have selected the different feature columns and implemented randomized search where we got about 91% accuracy

This code performs a grid search over different values of n_neighbors, weights, and p and identifies the best hyperparameters. It also standardizes the features using StandardScaler, which can be important for KNN algorithms.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import accuracy_score, classification_report
from scipy.stats import randint

# Assuming your dataset is in a DataFrame called 'df'
feature_columns = ['source', 'price', 'destination', 'temperatureMin', 'temperatureMax', "latitude", "longitude", "distance"]
target_variable = 'cab_type'

# Create feature matrix (X) and target variable (y)
X = df[feature_columns]
y = df[target_variable]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize KNN classifier
knn = KNeighborsClassifier()

# Define hyperparameter distributions for randomized search
param_dist = {
    'n_neighbors': randint(1, 20),  # Random integer values from 1 to 20
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}

# Perform randomized search with cross-validation
randomized_search = RandomizedSearchCV(knn, param_distributions=param_dist, n_iter=10, cv=5, scoring='accuracy', random_state=42)
randomized_search.fit(X_train, y_train)

# Get the best model from the search
best_model = randomized_search.best_estimator_

# Train the best model on the full training set
best_model.fit(X_train, y_train)

# Predict on the test set
y_pred = best_model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print best hyperparameters, accuracy, and classification report
print('Best Hyperparameters:', randomized_search.best_params_)
print(f'Accuracy: {accuracy}')
print('\nClassification Report:\n', classification_report(y_test, y_pred))
```

```
Best Hyperparameters: {'n_neighbors': 2, 'p': 2, 'weights': 'distance'}
Accuracy: 0.9120975579171761

Classification Report:
              precision    recall  f1-score   support

        Lyft       0.90      0.92      0.91     61339
        Uber       0.93      0.90      0.91     66257

    accuracy                           0.91    127596
   macro avg       0.91      0.91      0.91    127596
weighted avg       0.91      0.91      0.91    127596
```

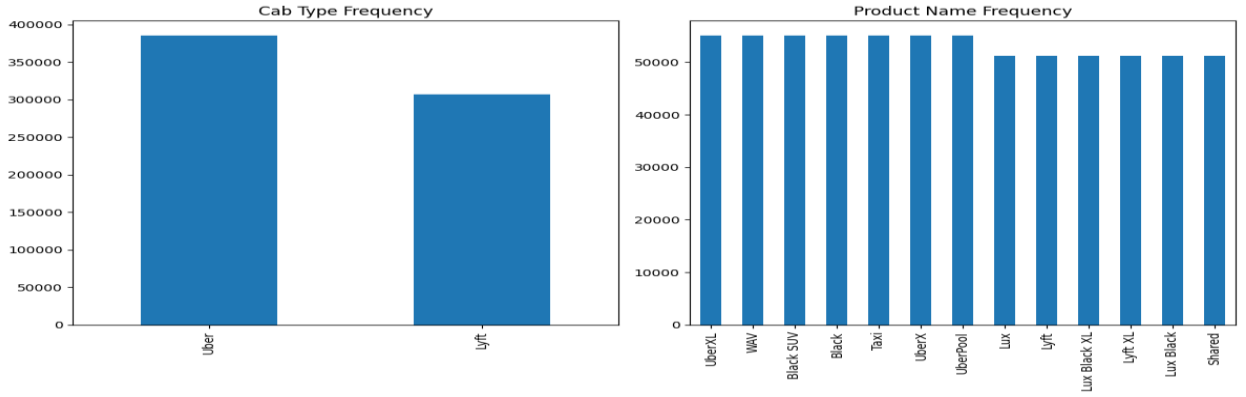
Plotting the graph results:

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

# Extract the results of the randomized search
results = pd.DataFrame(randomized_search.cv_results_)

# Create a heatmap to visualize the performance
heatmap_data = results.pivot_table(index='param_n_neighbors', columns='param_weights', values='mean_test_score')
sns.heatmap(heatmap_data, annot=True, cmap='viridis', fmt='.3f', cbar_kws={'label': 'Mean Test Score'})
plt.title('Randomized Search Results')
plt.xlabel('Weights')
plt.ylabel('Number of Neighbors')
plt.show()
```

Visualization Techniques Used:

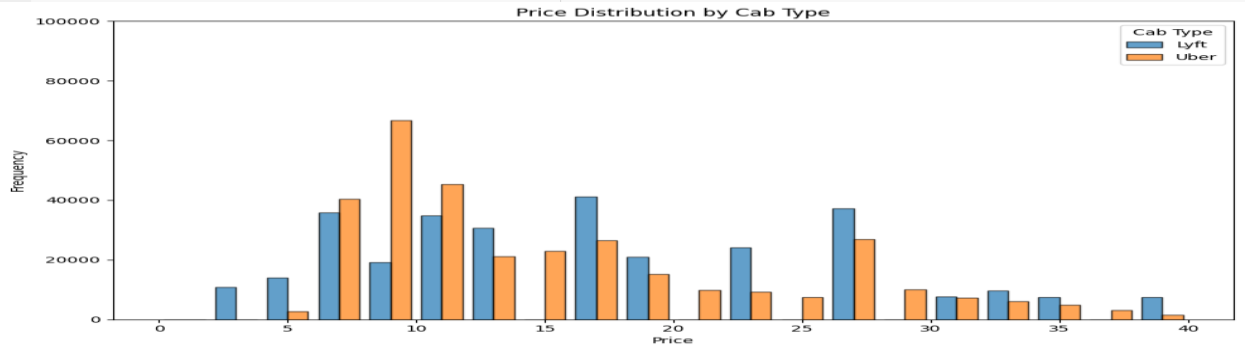
Bar Chart:

In [42]:
```
# Bar Plots
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
subset_df['cab_type'].value_counts().plot(kind='bar')
plt.title('Cab Type Frequency')
plt.subplot(1, 2, 2)
subset_df['name'].value_counts().plot(kind='bar')
plt.title('Product Name Frequency')
plt.tight_layout()
plt.show()
```



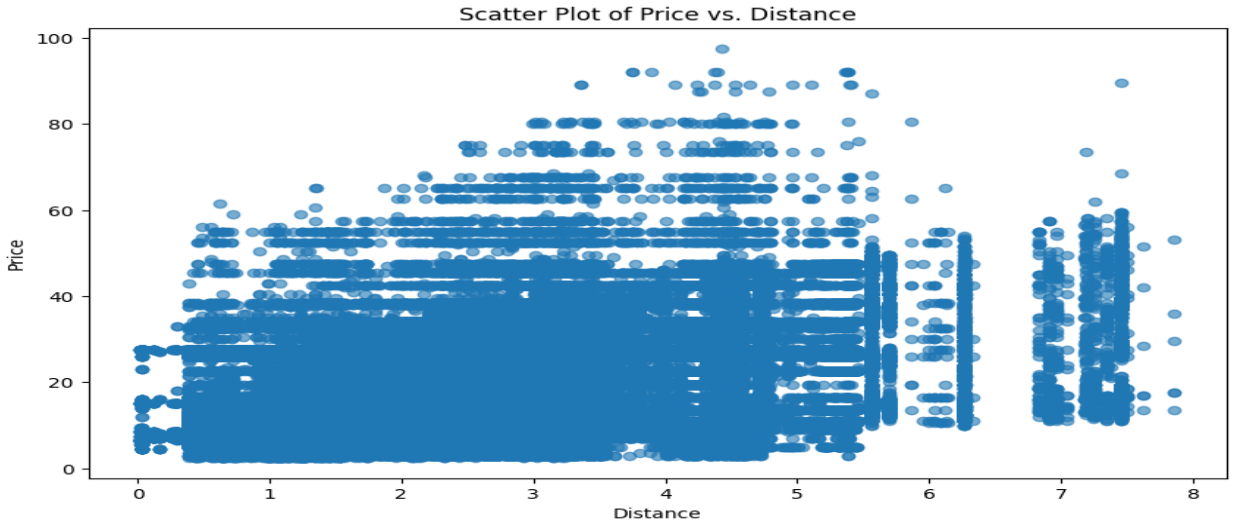
Histogram:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

# Define the data
cab_types = df['cab_type'].unique()

# Set the number of bins for the price histogram
num_bins = 20

# Create a list of data for each cab type
data = [df[(df['cab_type'] == cab_type) & (df['price'] >= 0) & (df['price'] <= 40)]['price'] for cab_type in cab_types]

# Plot the histograms with specified ranges
plt.hist(data, bins=num_bins, range=(0, 40), alpha=0.7, label=cab_types, edgecolor='black')

# Set the y-axis limit to 100,000
plt.ylim(0, 100000)

# Add labels and title
plt.title('Price Distribution by Cab Type')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.legend(title='Cab Type')

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



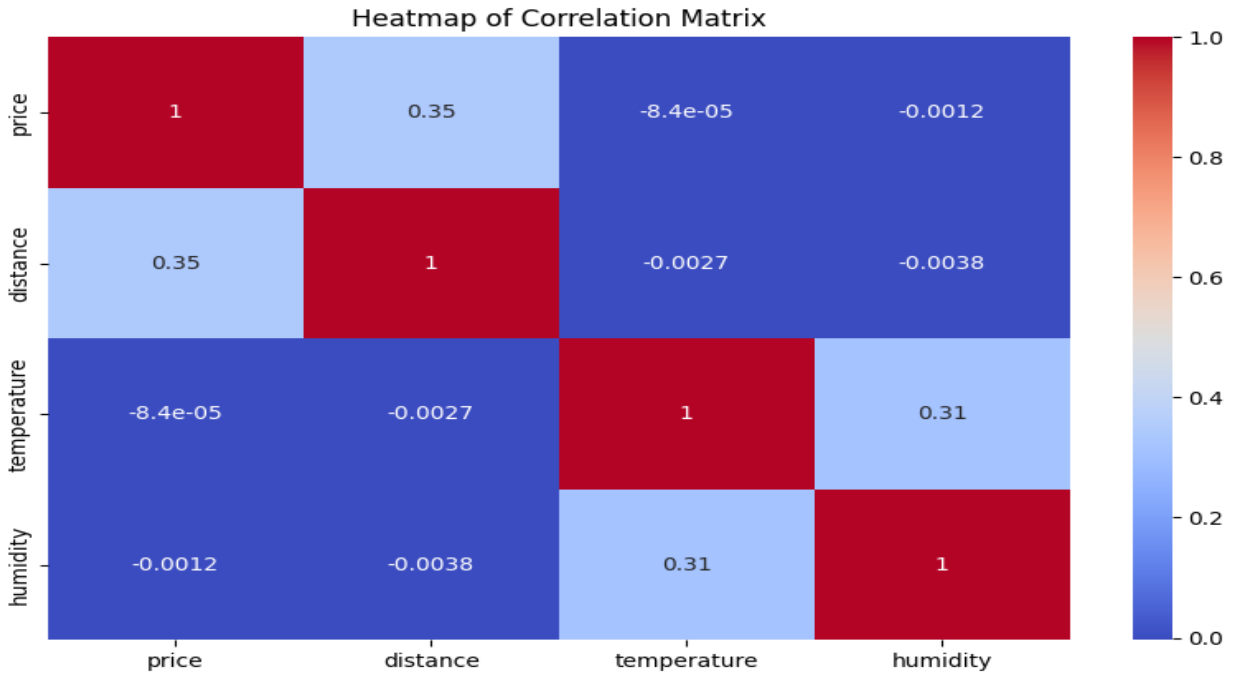
Scatter Plot:

```
#Scatter Plot

plt.figure(figsize=(10, 6))
plt.scatter(df['distance'], df['price'], alpha=0.6)
plt.title('Scatter Plot of Price vs. Distance')
plt.xlabel('Distance')
plt.ylabel('Price')
plt.show()
```



Correlation Matrix:

```
#Bivariant - Numerical & Numerical - Corelation matrix
numerical_attributes = ['price', 'distance', 'temperature', 'humidity']
correlation_matrix = df[numerical_attributes].corr()

plt.figure(figsize=(10, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Heatmap of Correlation Matrix')
plt.show()
```



Data Optimization Conclusion:

In the pursuit of enhancing prediction performance, a comprehensive optimization strategy was employed across various machine learning models, including Decision Trees, k Nearest Neighbors (kNN), Logistic Regression, Naïve Bayes, and ADA Boost. The powerful XG Boost algorithm sequentially combined weak learners, achieving an impressive 95% accuracy. Grid search was then employed for a Decision Tree model, elevating accuracy from 52% to 83%. Subsequently, randomized search for the KNN model resulted in a substantial accuracy increase from 50% to an impressive 91%. A noteworthy development includes the implementation of ADA Boost, yielding a commendable accuracy of 97.59%. The accompanying graph illustrates the importance of various elements in the decision-making process, showcasing feature prioritization. The X-axis, representing the index of features, aligns with their respective importance. This visual insight enhances our understanding of the ADA Boost algorithm's decision-making dynamics. The overall analysis encompassed model selection, preprocessing steps such as dummy encoding and feature scaling, and continuous monitoring for accuracy maintenance. In conclusion, selecting the final model and its parameters required a meticulous assessment of the data, task intricacies, and trade-offs between predictability,and simplicity. Continuous monitoring and updates were deemed essential to maintain model accuracy and relevance. The dynamic approach ensured above 80% accuracy across techniques, with ADA Boost standing out at 97.59%. This strategic optimization underscores the pivotal role of hyperparameter tuning and thoughtful model selection in achieving excellence in predicting sales for ride-sharing services like Uber and Lyft.