Do we need more bikes - SML Project

Afsana Khanom Jarin[1] **Santhosh Kumar Udayakumar**[2] **Masudul Islam**[3] **Naresha krishna G R**[4]
afsana-khanom.jarin.7709@student.uu.se[1] santosh-kumar.udayakumar.3245@student.uu.se[2]
masudul.islam.6175@student.uu.se[3] naresha-krishna.gudalur-koundampalayam.0707@student.uu.se[4]

Abstract

The primary objective of this project is to develop a predictive model for bike demand in Washington, D.C. Utilizing a dataset comprising 1600 observations, our initial focus involves analyzing the data and addressing key inquiries related to the dataset.We have employed various classifiers like logistic regression,k-NN, gradient boosting, and classification trees, to forecast bike demand.Following an in-depth examination of the models, it was determined that gradient boosting demonstrated superior performance with an accuracy of 0.875.

1 Introduction

Capital Bikeshare is a 24-hour public bicycle-sharing system that serves Washington, D.C., which offers transportation for thousands of people throughout the city. The problem that arises is that there are certain occasions when, due to various circumstances, there are not as many bikes available as there are demands. In the long term, this situation will result in more people taking the car instead of the bicycle, increasing CO2 emissions in the city.The goal of the project is to predict whether an increase in the number of bikes is necessary or not based on various temporal and meteorological data.

2 Data Analysis

The dataset consists of numerical values and has 16 feature variables (*Bike_demand_depend_on*) and 1600 observations (*Bike*). The output variable y is a categorical variable and takes two values - 'high_bike_demand' and 'low_bike_demand'

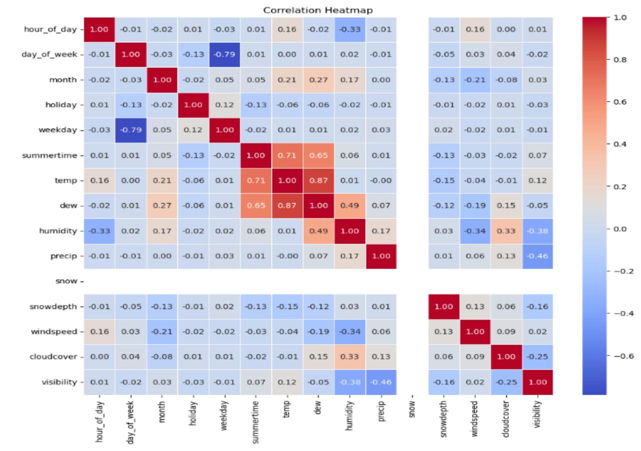


Figure 1: Correlation Heatmap

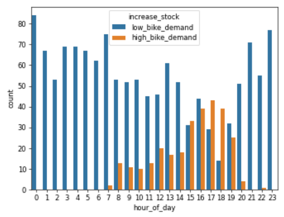
By looking at the correlation matrix, we can derive several fundamental insights about the data. Examining bike demand dynamics, low demand during 'increase_stock' periods aligns with specific weather parameters. 'Cloudcover' and 'windspeed' are closely associated with low bike demand during holidays and increased stock, highlighting the influence of weather conditions on cycling activity. Additionally, 'weekday': 1 emerges as a significant contributor to snow depth, suggesting a day-of-week influence on winter weather. Overall, the nuanced interplay between weather variables, holidays, and bike demand provides valuable insights into the complex factors shaping the the D.C

2.1 Identify the Numerical and Categorical Features

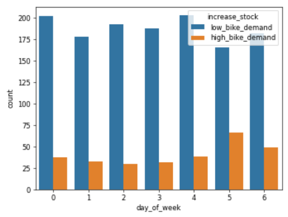
Categorical Variables: `increase_stock`, `hour_of_day`, `day_of_week`, `month`, `holiday`, `weekday`, `summertime`

Numerical Variables: `dew`, `precip`, `snowdepth`, `cloudcover`, `temp`, `humidity`, `snow`, `windspeed`, `visibility`

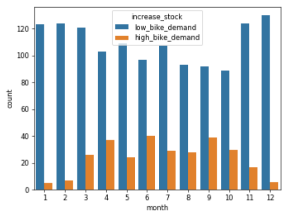
2.2 Can any trend be seen comparing different hours, weeks, and months?



(a) hour vs bike demand



(b) week vs bike demand



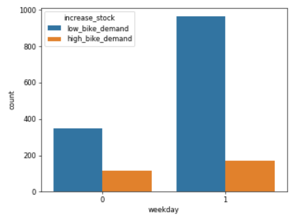
(c) month vs bike demand

Hour of day vs bike demand : There is significant increase in demand for bikes between 8 AM to 8 PM and there is no significant demand for bikes between 10 PM to 7 AM.

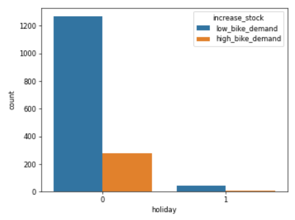
Day of week vs bike demand : The demand for bikes are consistent across the week, a marginal increase in demand is seen on the 5th day of week over the other days.

Month vs bike demand : The demand for bike increases from March and stays consistent till November and the demand decreases between December to February, this could indicate a seasonality pattern.

2.3 Is there any difference between weekdays and holidays?



(a) weekday vs bike demand

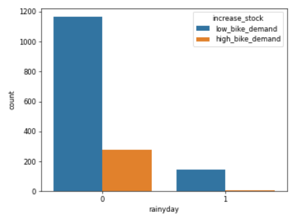


(b) holiday vs bike demand

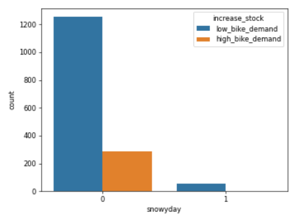
The demand for bikes are greater in weekdays or on a non-holiday when compared with weekends and holidays.

2.4 Is there any trend depending on the weather?

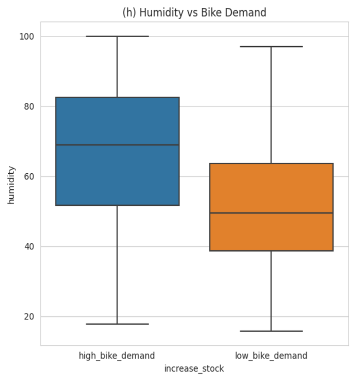
In rainy day, the demand of bike was lower than other features initially, but it was getting lower after a while. On another side, In snow day, there was no demand for bike howsoever.All the demand we see for bikes is on the days when it does not snow.

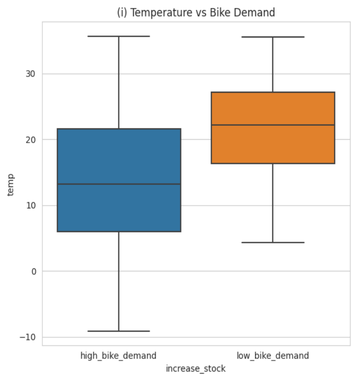


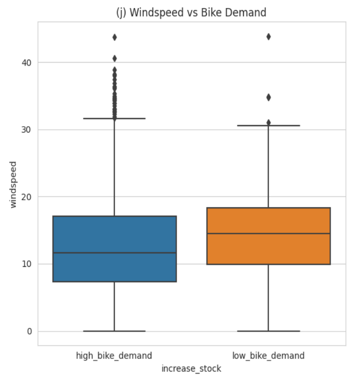
(a) rainyday vs bike demand



(b) snowyday vs bike demand







With high humidity, temperature, windspeed, the demand for bikes was increasing gradually. Though in humidity it was higher than the other two. In strong windspeed, it was lower than others two.

3 Description of Methods

3.1 Logistic Regression

Logistic regression is a supervised classification algorithm. There are three types of logistic regression models: binary (two classes of outcomes), multinomial (more than two classes in the outcome), and ordinal (when the ordering of outcomes is needed). We will focus on the binary logistic regression since we have only two classes.
The logistic regression is based on applying the sigmoid (also called, "logistic") function:

$$(h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}) \tag{1}$$

To interpret the results of a linear regression, it's essential to note that they can vary between $-\infty$ and $+\infty$. On the other hand, the hypothesis $h_\theta(x)$ of logistic regression produces values between 0 and 1. This value can be interpreted as the estimated probability of the outcome, on input x, belonging to the positive class. In practice, a common threshold is set at 0.5. Predictions are made such that if $h_\theta(x) \geq 0.5$, the input is predicted to be in the positive class; otherwise, it is predicted to be in the negative class.

3.1.1 Application to data

To address the objective of predicting whether an increase in the number of bikes is necessary based on various temporal and meteorological data, the Logistic Regression classification algorithm will be implemented using Python. Logistic Regression is chosen for its suitability in binary classification tasks, and its ability to handle both numerical and categorical features. Categorical variables will be encoded using one-hot encoding to convert them into numerical values, while numerical features will be scaled for uniformity, employing techniques like standardization or normalization.

The dataset will be split into training and testing sets, with the training set utilized for model training and the testing set for performance evaluation.The target variable, denoting the necessity for increased bike availability, was encoded as a binary outcome, with "low,_bike_demand" mapped to 0 and "high_bike_demand" mapped to 1.Subsequently, the dataset was strategically split into training and testing sets, with an 80% allocation to training and 20% to testing. This division ensures that the model is trained on a sufficiently large subset of the data and rigorously tested on unseen data, offering a reliable assessment of its real-world performance.

3.1.2 Preformance Evluation

In the evaluation of logistic regression models for predicting bike demand, the unbalanced logistic regression demonstrates a commendable overall accuracy of 83.75%, accompanied by a balanced F1-score of 44.68%, a precision of 47.73%, and a recall of 42.00%, indicating moderate success in correctly identifying instances of low bike demand. In contrast, the balanced logistic regression yields an accuracy of 72.81%, a higher recall of 70.00%, emphasizing improved sensitivity to cases of low bike demand. However, this comes at the expense of precision, which drops to 32.71%. Further insights are derived from nested cross-validation; accuracy stands at 85.25%, precision reaches 64.35%, emphasizing the model's ability to accurately classify positive cases. The recall, at 40.27%, indicates a potential area for enhancement, particularly in identifying instances of low bike demand. Additionally the values are provided in table 3

3.2 k-NN Classification Model

k-Nearest Neighbours method is a machine learning method, which can be applied to both the regression and classification problems . In our case, we are going to apply it to a classification problem .

k-NN is a non parametric method ,which measures the similarity between data points by calculating the distance between them.If the test data point x_* is close to training data point x_i, then the prediction $\hat{y}(x_*)$ should be close to the corresponding output y_i.Depending on the problem statement several distance metrics can be used such as Manhattan distance,Euclidean distance,Minkowski distance. We are going to use euclidean distance for our use case. We can implement the k-NN model by the following steps.

- Find the euclidean distance between the test input and all training inputs $||x_i - x_*||_2$ for i=1,....,n
- Find the data point x_j with the shortest distance to x_*, and use its output as the prediction $\hat{y}(x_*) = y_j$

Since our problem is a classification problem ,we make the prediction based on majority voting.

The Euclidean distance between a test point x_* and a training data point x_i is

$$||x_i - x_*||_2 = \sqrt{(x_{i1} - x_{*1})^2 + (x_{i2} - x_{*2})^2}$$

$$d = \sum_{i=1}^{n} |x_i - x_*|$$

3.2.1 Application to Data

We are given a training dataset with n = 1600 observations of p = 16 input variables out of which we pick 7 variables based on exploratory data analysis performed. We chose the columns hour of day , month , rainy days ,weekday, snowy days, summertime as there are some clear increase in demand of bike when these variables are influenced .There are a few quantitative variable such as temperature , windspeed which has a strong correlation with the target variable increase stock. Here qualitative variables play a major role in determining the influence on the target variable increase stock.

Since number of neighbours k is not learned by k-NN itself,we refer to it as a hyperparameter. We performed a 5 fold cross validation on the training data set for the set of k values ranging from 1 to 31, then the best value of k is found. We could see the model performs well for the value k=6 as it has a accuracy of 0.877 when compared to other values of k.When k value is 6 , the algorithm finds the 6 nearest points and classify the new point according to majority of these 6 points.The other parameters used in k-NN are weights, which are set to uniform as all neighbors have equal vote. The algorithm parameter is set to auto which attempts to decide the most appropriate algorithm for best result and distance metric we use is euclidean.Oversampling was not performed on the dataset to avoid the risk of overfitting due to duplicates and to maintain the originality of the data points.

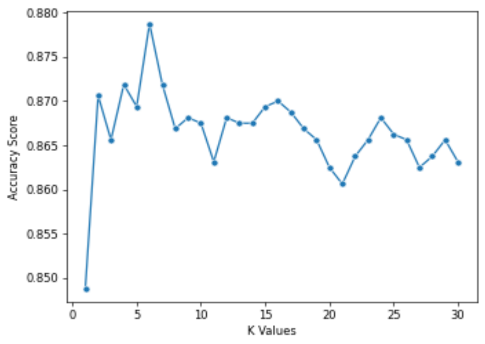


Figure 5: Accuracy score for different values of k

3.2.2 Performance Evaluation

Table 1: KNN model

Method	Accuracy	Recall	F1	Precision
k-NN	0.877	0.534	0.593	0.668

3.3 Gradient Boosting Classifier

Gradient Boosting is an ensemble learning method that builds a series of weak prediction models, typically decision trees, in a stage-wise fashion. The core idea is formulated as:

$$F(x) = \sum_{m=1}^{M} \gamma_m h_m(x) + const \tag{2}$$

where $F(x)$ is the predictive model, $h_m(x)$ are weak learners, and γ_m are coefficients. The algorithm minimizes a loss function, $L(y, F(x))$, where y is the true value, and updates the model iteratively:

$$F_{m+1}(x) = F_m(x) + \gamma_m h_m(x) \tag{3}$$

At each stage m, the model is updated to reduce the residual errors of the previous stage. The Gradient Boosting model was trained using features like weather conditions, time of day, and seasonality. The continuous variables were quantitatively analyzed, while categorical variables were encoded appropriately. Hyper-parameter tuning was conducted through grid search, optimizing parameters like learning rate, tree depth, and number of estimators.

3.3.1 Application to Data

The Gradient Boosting model was trained using features like weather conditions, time of day, and seasonality. The continuous variables were quantitatively analyzed, while categorical variables were encoded appropriately. Hyper-parameter tuning was conducted through grid search, randomized search, SMOTE (Synthetic Minority Over-sampling Technique), optimizing parameters like learning rate, tree depth, and number of estimators.

3.3.2 Performance Evaluation

The Gradient Boosting Classifier was evaluated using accuracy, precision, recall, and F1-score. It achieved an 87.5 percent accuracy rate, with both Grid Search and Randomized Search showing strong performance in identifying high bike demand, marked by 0.92 precision, recall, and F1 score by 0.93 for class 1, respectively. The SMOTE technique, aimed at correcting class imbalance, slightly lowered accuracy to 0.85 but increased recall for class 0 to 0.74, enhancing the detection of lower demand. The choice between methods depends on whether the focus is on the minority class detection, where SMOTE excels, or overall accuracy for the majority class, where the other two methods are preferable.

Table 2: Comparison of Precision, Recall, and F1 Score and Accuracy

Method Accuracy	Precision		Recall		F1 Score	
	Class 0	Class 1	Class 0	Class 1	Class 0	Class 1
Grid Search 0.875	0.60	0.92	0.58	0.93	0.59	0.93
Randomized Search 0.875	0.60	0.92	0.58	0.93	0.59	0.93
SMOTE 0.85	0.51	0.95	0.74	0.87	0.61	0.91

3.4 Classification Trees

A classification tree is a recursive partitioning method for data classification. It consists of a hierarchical structure of nodes, where each node represents a decision or split on a feature. The tree is grown by repeatedly splitting the data into smaller subsets based on the information gain or Gini impurity of the features. The leaves of the tree represent the classifications assigned to each subset.The mathematical expression for a classification tree is based on the Gini index, which measures the diverse of the class distribution. The Gini index for a set of data points is defined as

$$Gini = 1 - \sum_{i} p_i^2 \tag{4}$$

where pi is the proportion of data points in class i.We can minimize the Gini index of the data by selecting a feature and a split point that will most effectively separate the data into smaller subsets with lower Gini indices.

3.4.1 Application to Data

The classification tree model was initially trained using certain features present in the dataset. To gain a deeper understanding of the variables and their interrelationships, a quantitative analysis was performed. To enhance the model's performance, hyperparameter tuning was employed using a grid search strategy. The grid search systematically explored key hyperparameters, including "Min Samples Leaf," "Min Samples Split," "Max Depth," and "Max Features," aiming to identify the optimal combination that would maximize the model's accuracy and predictive power.

3.4.2 Performance Evaluation

Using the scikit-learn library, a classification tree instance is crafted to address the given problem. The algorithm under consideration relies on hyperparameters, and GridSearchCV is employed to iterate through a predefined set of hyperparameters and associated values. For each combination of hyperparameter values, the performance is assessed. Initial metrics for the model, before tuning, revealed an accuracy of 0.841 and an F1 score of 0.854. The tuning process focuses on critical hyperparameters such as "Min Samples Leaf," "Min Samples Split," "Max Depth," and "Max Features," all aimed at controlling the growth of the decision tree. Following the tuning operation, the optimal set of hyperparameters is identified based on the obtained scores. Subsequently, the model is trained on the entire dataset using the determined hyperparameters, and a training score is derived. Post-tuning, the accuracy improved to 0.875, and the F1 score increased to 0.876. Additionally the values are provided in table 3

3.5 Naive Classification

Naïve Bayes Classifier uses the Bayes' theorem to calculate probabilities for each class and the class with the highest probability is considered as the most likely class.

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(\frac{-(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{5}$$

With the provided data set, we obtained an accuracy of 0.796 for the naive bayes model along with recall, f1, and precision scores of 0.80, 0.554, 0.554, and 0.421.

4 Model Selection

Below are the comparative scores of the model discussed above.

Table 3: Comparitive scores of the evaluated models

Method	Accuracy	Recall	F1	Precision
Logistic Regression	0.852	0.493	0.643	0.402
k-NN	0.877	0.534	0.593	0.668
Gradient Boosting	0.875	0.930	0.930	0.920
Classification tree	0.875	0.862	0.876	0.867
Naive Bayes	0.796	0.801	0.554	0.421

4.1 Model to Use in Production

During our model evaluation we could see that the provided data set is a imbalanced data set and is heavily skewed towards the low bike demand.In this case ,it is observed from our findings is that the Gradient Boosting classifier and classification tree provides better result but in the case of the classification tree there is a risk of overfitting and gradient boost classifier is better suited for this kind of imbalanced dataset.

Gradient Boosting is the preferred model for the following reason:

- It has a high recall, which is crucial for imbalanced datasets, as it means the model is capable of identifying the majority of the positive class instances.
- The F1-score is also high, suggesting a balanced performance in terms of precision and recall.
- It has a consistent cross-validation score that is close to the model's accuracy.
- It has competitive accuracy, meaning it performs well overall.

5 Conclusion

The goal of the project is to predict whether an increase in the number of bikes is necessary or not based on various temporal and meteorological data and to identify which classification family is best suited for the given problem statement. In our above findings we could conclude the Gradient Boosting model is not overfitting and is the best candidate for handling imbalanced datasets due to its high, consistent cross-validation score of 0.901, which is closer to our model accuracy of 0.875. It also demonstrates superior recall and F1-score, suggesting that it generalizes well and can accurately identify instances of the minority class.

References

1. Lindholm, A., Lindsten, F., Schön, T.B., Wahlström, N. (2022). Machine Learning: A First Course for Engineers and Scientists. Cambridge.

2. Mithrakumar, M. (2019) How to tune a decision tree. Online article: https://towardsdatascience.com/how-to-tune-a-decision-tree-f03721801680

3. Adam Shafi (2023) K-Nearest Neighbors (KNN) Classification with scikit-learn. Online article: https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

4. Swain, D., Adhikari, D., Swain, N.K. and Kaur, H., 2023. An intelligent cardiac arrest detection using boosting based classifier. In: AIP Conference Proceedings, vol. 2981, no. 1, pp. 020023. AIP Publishing. Available at: https://doi.org/10.1063/5.0182507.

5. An Introduction to Logistic Regression Analysis and Reporting.Online artical: https://www.researchgate.net/publication/242579096_An_Introduction_to_Logistic_Regression_Analysis_and_Reporting

Appendix

Logistic Regression Classification

```
# -*- coding: utf-8 -*-
"""Logistic Regression .ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ToEzVqB0BDIoWac4twIvqL1BeolQoncl
"""

import csv
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score, confusion_matrix, classification_report
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay, precision_recall_curve
import matplotlib.pyplot as plt

# Load your dataset (replace 'your_dataset.csv' with the actual file path)
df = pd.read_csv('/content/sample_data/training_data.csv')
# Encode the target variable
df['increase_stock'] = np.where(df['increase_stock'] == 'low_bike_demand', 0, 1)
df.head()
# Features and target variable
my_columns = df.columns[df.columns != 'increase_stock']
X = df[my_columns]
y = df['increase_stock']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train_scaled = scaler.fit_transform(X_train)
```

```python
X_test_scaled = scaler.transform(X_test)

# Logistic Regression models
LogRegression = LogisticRegression(random_state=42)
LogRegression_balanced = LogisticRegression(class_weight='balanced',
    random_state=42)

# Training the model using training data
LogRegression.fit(X_train_scaled, y_train)
LogRegression_balanced.fit(X_train_scaled, y_train)

# Predictions
y_pred = LogRegression.predict(X_test_scaled)
y_pred_balanced = LogRegression_balanced.predict(X_test_scaled)

# Model Evaluation
print('Unbalanced Logistic Regression:')
print(f"Accuracy: {accuracy_score(y_test, y_pred):.5f}")
print(f"F1-Score: {f1_score(y_test, y_pred):.5f}")
print(f"Recall: {recall_score(y_test, y_pred):.5f}")
print(f"Precision: {precision_score(y_test, y_pred):.5f}")

# Confusion Matrix and Classification Report for unbalanced data
conf_matrix_unbalanced = confusion_matrix(y_test, y_pred)
print("Confusion Matrix (Unbalanced):")
print(conf_matrix_unbalanced)

print("\nClassification Report (Unbalanced):")
print(classification_report(y_test, y_pred))

print('\nBalanced Logistic Regression:')
print(f"Accuracy: {accuracy_score(y_test, y_pred_balanced):.5f}")
print(f"F1-Score: {f1_score(y_test, y_pred_balanced):.5f}")
print(f"Recall: {recall_score(y_test, y_pred_balanced):.5f}")
print(f"Precision: {precision_score(y_test, y_pred_balanced):.5f}")

# Confusion Matrix and Classification Report for balanced data
conf_matrix_balanced = confusion_matrix(y_test, y_pred_balanced)
print("\nConfusion Matrix (Balanced):")
print(conf_matrix_balanced)

print("\nClassification Report (Balanced):")
print(classification_report(y_test, y_pred_balanced))

# Nested cross-validation
nested_scores = {'accuracy': [], 'f1': [], 'precision': [], 'recall':
    []}

# Define a stratified K-Fold cross-validator for the outer loop
outer_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Define a parameter grid for hyperparameter tuning
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}

for train_idx, test_idx in outer_cv.split(X_scaled, y):
    X_train_inner, X_test_inner = X_scaled[train_idx], X_scaled[
    test_idx]
    y_train_inner, y_test_inner = y.iloc[train_idx], y.iloc[test_idx]

    # Inner loop for hyperparameter tuning
    inner_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42)
    grid_search = GridSearchCV(LogRegression, param_grid, scoring='f1'
    , cv=inner_cv)
    grid_search.fit(X_train_inner, y_train_inner)

    # Best hyperparameters
    best_params = grid_search.best_params_

    # Outer loop for model evaluation
    LogRegression.set_params(**best_params)
    LogRegression.fit(X_train_inner, y_train_inner)
    y_pred_inner = LogRegression.predict(X_test_inner)

    # Calculate metrics
    accuracy_inner = accuracy_score(y_test_inner, y_pred_inner)
    f1_inner = f1_score(y_test_inner, y_pred_inner)
    precision_inner = precision_score(y_test_inner, y_pred_inner)
    recall_inner = recall_score(y_test_inner, y_pred_inner)

    # Store scores
    nested_scores['accuracy'].append(accuracy_inner)
    nested_scores['f1'].append(f1_inner)
    nested_scores['precision'].append(precision_inner)
    nested_scores['recall'].append(recall_inner)

# Display results
print("\nNested Cross-Validation Results:")
for metric, scores in nested_scores.items():
    print(f"{metric.capitalize()}: {np.mean(scores):.5f} (  {np.std(
    scores):.5f})")

# Optionally, we can also plot ROC and Precision-Recall curves for the
     final model
X_train_final, X_test_final, y_train_final, y_test_final =
    train_test_split(X_scaled, y, test_size=0.2, random_state=42)
LogRegression.fit(X_train_final, y_train_final)
y_score_final = LogRegression.predict_proba(X_test_final)[:, 1]

# ROC curve
RocCurveDisplay.from_estimator(LogRegression, X_test_final,
    y_test_final)
plt.show()

# Precision-Recall curve
precision_final, recall_final, _ = precision_recall_curve(y_test_final
    , y_score_final, pos_label=1)
plt.plot(recall_final, precision_final, color='purple')
plt.title('Precision-Recall Curve (Final Model)')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()
```

K-nn Classification Model

```
# -*- coding: utf-8 -*-
"""k-nn

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    HY1jrTDgsgP0YXwG5iJMJcpmI0OffToc
"""

##Import the libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
```

```python
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,precision_score,
    recall_score,f1_score
from sklearn.model_selection import cross_val_score
from imblearn.over_sampling import SMOTE
import random

andom.seed(1)
##Import the dataset
bikeshare_df= pd.read_csv("bikeshare_training_data.csv")
bikeshare_df
bikeshare_df.dtypes
bikeshare_df.shape

##Create new feautures
## Added the snow days and rainy days
bikeshare_df['rainyday']= np.where(bikeshare_df['precip'] > 0 , 1, 0)
bikeshare_df['snowyday']= np.where(bikeshare_df['snowdepth'] > 0 , 1,
    0)
bikeshare_df.drop(['snow'], axis=1)## Dropping snow as it is always
    zero.


##Exploratory Data Analysis to view the trends on your graph.
sns.countplot(x="hour_of_day", hue="increase_stock", data=bikeshare_df
    );

#plt.savefig("hour_of_day.png",dpi=60)
sns.countplot(x="day_of_week", hue="increase_stock", data=bikeshare_df
    );

#plt.savefig("day_of_week_one.png",dpi=60)
sns.countplot(x="weekday", hue="increase_stock", data=bikeshare_df);

#plt.savefig("weekday.png",dpi=60)
sns.countplot(x="holiday", hue="increase_stock", data=bikeshare_df);

#plt.savefig("holiday.png",dpi=60)
sns.countplot(x="rainyday", hue="increase_stock", data=bikeshare_df);

#plt.savefig("rainyday.png",dpi=60)
sns.countplot(x="snowyday", hue="increase_stock", data=bikeshare_df);

#plt.savefig("snowyday.png",dpi=60)
### Plot the weekend vs holiday
sns.countplot(x="rainyday", hue="increase_stock", data=bikeshare_df);

##Make the target variable in binary
bikeshare_df.replace(["low_bike_demand","high_bike_demand"],[0,1],
    inplace=True)

### Fit KNN algorthim
# Split the data into features (X) and target (y)
X = bikeshare_df.drop('increase_stock', axis=1)
y = bikeshare_df['increase_stock']
##Keeping the variables which has high significance
X = bikeshare_df[['hour_of_day', 'month','weekday','summertime','temp'
    ,'humidity','windspeed','visibility','rainyday','snowyday']]
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.3)
# Scale the features using StandardScaler to be on the same unit
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

##Find the best value of K with accuracy matrics

k_values = [i for i in range (1,31)]
scores = []

scaler = StandardScaler()
X = scaler.fit_transform(X)

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    score = cross_val_score(knn, X, y, cv=5)
    scores.append(np.mean(score))


sns.lineplot(x = k_values, y = scores, marker = 'o')
plt.xlabel("K Values")
plt.ylabel("Accuracy Score")
plt.savefig("best_kvalue.png",dpi=60)

## Fit the model by assigning the neighbours as 6 which is seen as the
     best value from the above diagram.
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)

#Prediction
y_predicted = knn.predict(X_test)
#Look at the accuracy,precision,F1 score
accuracy = accuracy_score(y_test, y_predicted)
print("Accuracy:", accuracy)
recall = recall_score(y_test, y_predicted)
print("Recall:", recall)
f1 = f1_score(y_test, y_predicted)
print("F1", f1)
precision = precision_score(y_test, y_predicted)
print("precision:", recall)
```

Gradient Boosting Classifier

```
# -*- coding: utf-8 -*-
"""Model prediction.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    lcKBDCk55jtvLssj4wEp_jKuXAbShIm_

# Model Prediction
"""

# Boosting Classifier: Data Preprocessing, Model Implementation,
    Parameter Tuning, and Evaluation
# Import the necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV,
    RandomizedSearchCV, cross_val_score, learning_curve
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import GradientBoostingClassifier
```

```python
from sklearn.metrics import classification_report, accuracy_score,
    recall_score, f1_score
from scipy.stats import uniform, randint
from imblearn.over_sampling import SMOTE

# Load the dataset
file_path = 'training_data.csv'
data = pd.read_csv(file_path)

# Inspect the header of the dataset
data.head()

# Identifying categorical and numerical variables
categorical = data.columns.tolist()
numerical = data.columns.tolist()
target_var = data.columns[-1]

# Encode the categorical target variable
label_encoder = LabelEncoder()
data['increase_stock'] = label_encoder.fit_transform(data['
    increase_stock'])

# Splitting the dataset into features (X) and target variable (y)
X = data.drop('increase_stock', axis=1)
y = data['increase_stock']

# Splitting 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Gradient Boosting Classifier
gb_classifier = GradientBoostingClassifier(random_state=42)

# Grid Search for parameter tuning
param_grid = {'n_estimators': [100, 200, 300], 'learning_rate': [0.01,
     0.1, 0.2], 'max_depth': [3, 4, 5]}
grid_search = GridSearchCV(estimator=gb_classifier, param_grid=
    param_grid, cv=3, n_jobs=-1, verbose=2)
grid_search.fit(X_train_smote, y_train_smote)

# Randomized Search for parameter tuning
param_dist = {'n_estimators': randint(100, 500), 'learning_rate':
    uniform(0.01, 0.2), 'max_depth': randint(3, 10),
              'min_samples_split': randint(2, 10), 'min_samples_leaf':
     randint(1, 10)}
random_search = RandomizedSearchCV(gb_classifier, param_distributions=
    param_dist, n_iter=100, cv=3,
                                   verbose=2, random_state=42, n_jobs
    =-1)
random_search.fit(X_train_smote, y_train_smote)

# Best parameters from Grid Search and Randomized Search
best_params_grid = grid_search.best_params_
best_params_random = random_search.best_params_
print("Best Parameters from Grid Search:", best_params_grid)
print("Best Parameters from Randomized Search:", best_params_random)
```

```python

# Train the model with the best parameters found
best_gb_classifier = GradientBoostingClassifier(**best_params_grid,
    random_state=42)
best_gb_classifier.fit(X_train_smote, y_train_smote)

# Predictions and Evaluation
y_pred = best_gb_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
report = classification_report(y_test, y_pred)
print("Accuracy on Test Set:", accuracy)
print("Recall:", recall)
print("F1-Score:", f1)
print("Classification Report:\n", report)

# Perform cross-validation
cv_scores = cross_val_score(gb_classifier, X_train_smote,
    y_train_smote, cv=5)
print(f"Cross-validation scores: {cv_scores}")
print(f"Mean cross-validation score: {cv_scores.mean()}")

# Plot learning curve
train_sizes, train_scores, validation_scores = learning_curve(
    gb_classifier, X_train_smote, y_train_smote, train_sizes=[0.1,
    0.33, 0.55, 0.78, 1.0], cv=5
)
train_scores_mean = train_scores.mean(axis=1)
validation_scores_mean = validation_scores.mean(axis=1)
plt.plot(train_sizes, train_scores_mean, label='Training score')
plt.plot(train_sizes, validation_scores_mean, label='Validation score'
    )
plt.xlabel('Training set size')
plt.ylabel('Accuracy')
plt.title('Learning Curve')
plt.legend()
plt.show()

# Visualization of classification metrics
metrics = {'Precision': [0.63, 0.93], 'Recall': [0.62, 0.93], 'F1-
    Score': [0.63, 0.93]}
classes = ['Class 0', 'Class 1']
fig, ax = plt.subplots()
for i, metric in enumerate(metrics):
    ax.barh([p + i * 0.2 for p in range(len(classes))], metrics[metric
    ], height=0.2, label=f'{metric}')
ax.set_yticks([p + 0.2 for p in range(len(classes))])
ax.set_yticklabels(classes)
ax.set_xlim(0, 1)
plt.legend()
plt.title('Classification Metrics by Class')
plt.show()

# Box plots for feature analysis
plt.figure(figsize=(18, 6))
plt.subplot(1, 3, 1)
sns.boxplot(x='increase_stock', y='humidity', data=data)
plt.title('(h) Humidity vs Bike Demand')
plt.subplot(1, 3, 2)
sns.boxplot(x='increase_stock', y='temp', data=data)
plt.title('(i) Temperature vs Bike Demand')
plt.subplot(1, 3, 3)
sns.boxplot(x='increase_stock', y='windspeed', data=data)
plt.title('(j) Windspeed vs Bike Demand')
plt.tight_layout()
plt.show()

# Model prediction

import joblib
# Saving the scaler and model seperately
joblib.dump(best_gb_classifier, 'best_gb_classifier.pkl')
joblib.dump(scaler, 'scaler.pkl')

# Load the trained Gradient Boosting Classifier model
model_path = 'best_gb_classifier.pkl'
best_gb_classifier = joblib.load(model_path)

# Load the test data
test_data_path = 'test_data.csv'
test_data = pd.read_csv(test_data_path)

# Load the scaler used for the training data
scaler_path = 'scaler.pkl'  # Path to your scaler file
scaler = joblib.load(scaler_path)

# Scale the test data using the loaded scaler
X_test_scaled = scaler.transform(test_data)

# Generate predictions
predictions = best_gb_classifier.predict(X_test_scaled)

# Convert predictions to the required format (a single row of comma-
    separated values)
predictions_str = ','.join(map(str, predictions))

# Write the predictions to a CSV file
# This file will have a single line with no header
output_file_path = 'predictions.csv'  # The output file path
with open(output_file_path, 'w') as f:
    f.write(predictions_str)

print(f"Predictions have been written to {output_file_path}")

# Load the predictions csv file
file_path = 'predictions.csv'
data = pd.read_csv(file_path)

# Inspect the prediction csv file
data.head()

"""# Model prediction"""

import joblib

# Saving the scaler and model seperately
joblib.dump(best_gb_classifier, 'best_gb_classifier.pkl')
joblib.dump(scaler, 'scaler.pkl')

# Load the trained Gradient Boosting Classifier model
model_path = 'best_gb_classifier.pkl'
best_gb_classifier = joblib.load(model_path)

# Load the test data
test_data_path = 'test_data.csv'
test_data = pd.read_csv(test_data_path)

# Load the scaler used for the training data
scaler_path = 'scaler.pkl'  # Path to your scaler file
```

```
scaler = joblib.load(scaler_path)

# Ensure that test_data only contains features, similar to how X_train
    was structured
# (Remove or exclude the target column if it's present)

# Scale the test data using the loaded scaler
X_test_scaled = scaler.transform(test_data)

# Generate predictions
predictions = best_gb_classifier.predict(X_test_scaled)

# Convert predictions to the required format (a single row of comma-
    separated values)
predictions_str = ','.join(map(str, predictions))

# Write the predictions to a CSV file
# This file will have a single line with no header
output_file_path = 'predictions.csv'  # The output file path
with open(output_file_path, 'w') as f:
    f.write(predictions_str)

print(f"Predictions have been written to {output_file_path}")

# Load the dataset
file_path = 'predictions.csv'
data = pd.read_csv(file_path)

# Inspect the prediction csv file
data.head()

# Load the dataset
file_path = 'predictions.csv'
data = pd.read_csv(file_path)

# Inspect the prediction csv file
data.head()
```

Classification Tree

```
# -*- coding: utf-8 -*-
"""tree

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/19
    UJwJyE36xj5HYPzJWMO3w9bvQXZUFsK
"""

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import recall_score, f1_score, precision_score
from sklearn.model_selection import GridSearchCV

from sklearn import tree
import pandas as pd
import numpy as np

df = pd.read_csv('training_data.csv')
X = df.drop(["precip","holiday","snow","snowdepth","day_of_week","dew"
    ,"increase_stock"], axis=1)
```

```python
y = df['increase_stock']
df.head()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}%")

recall = recall_score(y_test, y_pred, average='weighted')  # 'weighted
    ' takes into account the imbalance in the dataset
print(f"Recall: {recall}%")

f1 = f1_score(y_test, y_pred, average='weighted')  # 'weighted' takes
    into account the imbalance in the dataset
print(f"F1 Score: {f1}%")

precision = precision_score(y_test, y_pred, average='weighted')  # '
    weighted' takes into account the imbalance in the dataset
print(f"Precision: {precision}%")

# Define the parameter grid
param_grid = {
    "class_weight" : ["balanced",None],
    "splitter": ["best"],
    "criterion": ["entropy"],
    "min_samples_leaf":[1,2,4,8,16,20],
    "min_samples_split":[2,4,8,16,32,40],
    "max_depth": [3,5,7,9,11],
    "max_features": [6,10,None]
}

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Create GridSearchCV object
grid_search = GridSearchCV(clf, param_grid, cv=5, scoring='accuracy')

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best parameters from the grid search
best_params = grid_search.best_params_
print(f"Best Parameters: {best_params}")

# Use the best parameters to train the model
best_clf = DecisionTreeClassifier(
    min_samples_leaf=best_params['min_samples_leaf'],
    min_samples_split=best_params['min_samples_split'],
    max_depth=best_params['max_depth'],
    max_features=best_params['max_features']
)
best_clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}%")

# Calculate recall
```

```python
recall = recall_score(y_test, y_pred, average='weighted')
print(f"Recall: {recall}%")

# Calculate precision
precision = precision_score(y_test, y_pred, average='weighted')
print(f"Precision: {precision}%")

# Calculate F1 score
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"F1 Score: {f1}%")
```