Do we need more bikes?
Project in Statistical Machine Learning

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

In this project, we aim to create a classification model that can predict whether an increase in the number of bikes is needed in Washington D.C. on a specific temporal and meteorogical conditions. We will compare various classification models which consists of Logistic Regression, Linear Discriminant Analysis (LDA), K-Nearest Neighbors (KNN), Random Forest, and Gradient Boosting. The models are evaluated using Accuracy, Precision, Recall, and F1-Score metrics with 10-Fold Cross Validation. The results indicate that the Random Forest model outperforms the other models in all evaluation metrics, with an accuracy of 94.44%, recall of 94.99%, precision of 94.00%, and F1-score of 94.47%. This suggests that Random Forest is the most effective model for predicting bike demand in this context.

Number of group member: **4**

1 Problem Description

Capital Bikeshare is a 24-hour public bicycle-sharing system that serves Washington, D.C., and 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. To tackle this situation, the District Department of Transportation in the city wants to know if at certain hours an increase in the number of bikes available will be necessary.

In this Project, we aim to analyze whether the increase in the number of bikes is necessary or not based on the various temporal and meteorogical data provided in the dataset.

2 Data Analysis

The Training Dataset training.csv consists of 1600 randomly selected observation over the period of three years in the city of Washington D.C. The dataset contains 16 features and 1 target variable. The features are: *hour_of_day*, *day_of_week*, *month*, *holiday*, *weekday*, *summertime*, *temp*, *dew*, *humidity*, *precip*, *snow*, *snow_depth*, *windspeed*, *cloudcover*, and *visibility*. And the target variable is *increase_stock*.

2.1 Variable Types and Processing

The target variable *increase_stock* indicates whether an increase in the number of bikes is needed at a particular hour. the value *'low_bike_demand'* indicates that no increase is needed, while *'high_bike_demand'* indicates that an increase is necessary. For the analysis, we will convert these categorical values into binary numerical values, where *'low_bike_demand'* is represented as 0 and *'high_bike_demand'* as 1. Since the target variable is binary, this problem can be treated as a binary classification task.

For Binary features such as *holiday*, *weekday*, and *summertime*, they will be counted as categorical variables with values 0 and 1.

The Features *temp*, *dew*, *humidity*, *precip*, *snow*, *snow_depth*, *windspeed*, *cloudcover*, and *visibility* will be treated as numerical variables as they represent continuous measurements.

Regarding Ordinal features such as *hour_of_day*, *day_of_week*, and *month*, special attention is needed. For these features, we can't simply treat them as numerical values due to their cyclical nature. For example, after hour 23 comes hour 0 again. However, treating them as categorical variables may lead to loss of information regarding their order and cyclical patterns. Therefore, there are several possible approaches to handle them:

- One-Hot Encoding: Convert each of these features into multiple binary features, each representing a specific category. For example, *hour_of_day* would be converted into 24 binary features.
- Cyclical Transformation: Transform these features using sine and cosine functions to capture their cyclical nature. For example, for *hour_of_day*, we can create two new features:

$$\text{hour\_sin} = \sin\left(2\pi \cdot \frac{\text{hour\_of\_day}}{24}\right) \text{[?]}$$

$$\text{hour\_cos} = \cos\left(2\pi \cdot \frac{\text{hour\_of\_day}}{24}\right) \text{[?]}$$

Since One-Hot Encoding wouldn't effectively capture the cyclical nature of these features and may lead to high dimensionality, for this analysis, we will use the Cyclical Transformation approach to handle these ordinal features. [**?**]

After processing, the dataset will consist of 18 features and 1 target variable. Those features are shown in Table **??**.

Table 1: Processed Features in the Dataset

Feature	**Type**	**Description**
hour_sin	Numerical	Sine transformation of hour of the day
hour_cos	Numerical	Cosine transformation of hour of the day
day_sin	Numerical	Sine transformation of day of the week
day_cos	Numerical	Cosine transformation of day of the week
month_sin	Numerical	Sine transformation of month of the year
month_cos	Numerical	Cosine transformation of month of the year
holiday	Binary / Categorical	Whether the day is a holiday or not (0 or 1)
weekday	Binary / Categorical	Whether the day is a weekday or not (0 or 1)
summertime	Binary / Categorical	Whether the day is in the summer time period or not (0 or 1)
temp	Numerical	Temperature in Celsius
dew	Numerical	Dew point temperature in Celsius
humidity	Numerical	Relative Humidity in percentage
precip	Numerical	Precipitation in mm
snow	Numerical	Amount of snow in the last hour in cm
snow_depth	Numerical	Accumulated snow depth in cm
windspeed	Numerical	Wind speed in km/h
cloudcover	Numerical	Percentage of cloud cover
visibility	Numerical	Distance in km at which objects or landmarks can be clearly seen and identified
increase_stock (Target)	Binary / Categorical	Whether an increase in bike stock is needed (0 or 1)

2.2 Exploratory Data Analysis

For the initial stage, we will perform Exploratory Data Analysis (EDA) to understand the distribution and trends that arises in the dataset. Including which features are more correlated with the target variable *increase_stock*.

The feature *snow* only contains zero values in all observations, therefore it will be removed from the dataset as it doesn't provide any useful information for the analysis. Upon analyzing the dataset, we found that there are no missing values in any of the features or the target variable. Therefore, no handling is required.

Using Pearson correlation coefficient, we found correlation values between each feature and the target variable as shown in Table **??**.

Table 2: Ordered Correlation between Features and Target Variable

Feature	**Correlation Coefficient**
hour_of_day_cos	-0.339960
temp	0.336981
humidity	-0.308726
hour_of_day_sin	-0.308121
summertime	0.216052
month_cos	-0.169059
dew	0.132663
weekday	-0.116446
visibility	0.113443
windspeed	0.096011
month_sin	-0.092078
day_of_week_sin	-0.088152
precip	-0.059304
snowdepth	-0.047526
cloudcover	-0.045534
day_of_week_cos	-0.031473
holiday	-0.004909

As the table suggests, the feature *hour_of_day_cos* has the highest positive correlation with the target variable *increase_stock*, indicating that the time of day plays a significant role in determining whether an increase in bike stock is needed. On the other hand, the feature *holiday* has the lowest correlation with the target variable, suggesting that whether a day is a holiday or not has minimal impact on bike demand.

2.3 Imbalance in the Dataset

Upon analyzing the target variable *increase_stock*, we found that there is 1312 instances of class 0 (low bike demand) and 288 instances of class 1 (high bike demand). This indicates a significant class imbalance in the dataset [**?**], with class 0 being the majority class.

To address this class imbalance, we will employ the use of Synthetic Minority Over-sampling Technique (SMOTE). SMOTE works by generating synthetic samples for the minority class (class 1 in this case) by interpolating between existing minority class instances. This helps to balance the class distribution and provides the model with more representative samples of the minority class during training. [**?**]

The interpolation is done by selecting a minority class instance and finding its k-nearest neighbors. A synthetic sample is then created by randomly selecting one of the neighbors and interpolating between the two instances. This process is repeated until the desired balance between the classes is achieved.

3 Models and Methods

In this experiment, we will compare various classification models to determine which one performs best for predicting whether an increase in bike stock is needed using the provided dataset. The models we will consider includes Logistic Regression, Linear Discriminant Analysis (LDA), K-Nearest Neighbors (KNN), Random Forest, and Gradient Boosting.

For each model, we will perform hyperparameter tuning using techniques such as Grid Search or Random Search combined with cross-validation to find the optimal set of hyperparameters that yield the best performance on the validation set.

3.1 Benchmark Model

As the benchmark model, we will use a naive model that predict each instance using stratified random sampling based on the training set's class distribution. This means that for each instance, the model will randomly assign a class label (0 or 1) based on the proportion of each class in the training data [**?**]. This will provide a baseline accuracy to compare the performance of more sophisticated models.

3.2 Evaluation Metrics

To evaluate the performance of each classification model, we will use several metrics including Accuracy, Precision, Recall, and F1-Score. These metrics will provide a comprehensive understanding of how well each model performs in predicting the target variable. [**?**]

We will also use K-Fold Cross Validation to ensure that our evaluation metrics are robust and not overly dependent on a particular train-test split.

K-Fold Cross Validation involves dividing the dataset into K subsets, using one of the subsets as the test set and the remaining K-1 subsets as the training set. This process is repeated K times, with each subset used as the test set once. The final evaluation metrics are then averaged over all K iterations to provide a more reliable estimate of model performance. For this experiment, we will use K=10. [**?**]

3.3 Logistic Regression

Logistic Regression is a linear model used for binary classification tasks. It models the probability of the target variable being in a particular class using the logistic function. The model estimates the coefficients for each feature, which represent the impact of each feature on the log-odds of the target variable [**?**].

In Logistic Regression, we use the sigmoid function to map the linear combination of features to a probability value between 0 and 1. With the formulation:

$$P(Y=1|X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+...+\beta_n X_n)}}[?]$$

Where $P(Y=1|X)$ is the probability of the target variable being in class 1 given the features X, β_0 is the intercept, and $\beta_1, \beta_2, ..., \beta_n$ are the coefficients for each feature $X_1, X_2, ..., X_n$.

The parameters of the model are estimated using Maximum Likelihood Estimation (MLE), which finds the set of coefficients that maximize the likelihood of the observed data given the model.

3.4 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a classification method that finds a linear combination of features that best separates the classes. [**?**].

LDA is derived from the probabilistic model which models the class-conditional distributions of the data P(Xly=k) for each class k [**?**]. Predictions are made by applying Bayes theorem for each training sample $x \in \mathbb{R}^d$:

$$P(y=k|x) = \frac{P(x|y=k)P(y=k)}{P(x)}[?]$$

Then we select the class with the highest posterior probability. For LDA, P(X|y=k) is modeled as a multivariate Gaussian distribution with density function as follows:

$$P(x|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\right) [?]$$

where μ_k is the mean vector of class k, and Σ is the shared covariance matrix across all classes, and d is the number of features.

In LDA, we assume that the covariance matrices of all classes are equal, i.e., $\Sigma_k = \Sigma$ for all k. This reduces the log posterior to:

$$\log P(y=k|x) = -\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k) + \log P(y=k) + Constant[?]$$

which can be simplified to:

$$\log P(y=k|x) = w_k^T x + w_{k0} + Constant[?]$$

where $w_k = \Sigma^{-1}\mu_k$ and $w_{k0} = -\frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log P(y=k)$. [**?**]

3.5 K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a non-parametric classification method that classifies new instances based on the majority class of their K nearest neighbors in the feature space. The distance metric used to determine the nearest neighbors can be Euclidean distance, Manhattan distance, or other distance measures. [**?**]

KNN algorithm are usually used for unsupervised learning tasks such as clustering. However, it can also be adapted for supervised learning tasks by using the labels of the nearest neighbors to make predictions.

For Supervised learning tasks, KNN works using these following steps:

- Choose the number of neighbors K.
- For each new instance to be classified, calculate the distance between the new instance and all instances in the training dataset.
- Identify the K nearest neighbors based on the calculated distances.
- Determine the majority class among the K nearest neighbors.
- Assign the majority class as the predicted class for the new instance.

For the distance metric, we will use Manhattan distance, which is defined as:

$$d(p,q) = \sum_{i=1}^{n} |p_i - q_i| [?]$$

where p and q are two instances in the feature space, and n is the number of features.

3.6 Random Forest

Random Forest is an ensemble learning method that consists of multiple decision trees. Each tree is trained on a random subset of the training data and a random subset of features. The final prediction is made by aggregating the predictions from all trees. The method to aggregate the results in this experiment will use majority voting. [**?**]

Decision trees in principle work by recursively splitting the data based on feature values to create branches that lead to leaf nodes representing class labels. The splits are chosen based on the feature that maximizes the information gain or minimizes the impurity at each node. For the metric to measure impurity, we will use Entropy, which is defined as:

$$H(X) = -\sum_{i=1}^{c} p_i \log_2(p_i) [?]$$

where p_i is the proportion of instances belonging to class i in the node, and c is the number of classes.

Decision trees can be prone to overfitting, especially when they are deep and complex. To mitigate this, we can use Random Forest.

Random Forest Algorithm, works using these following steps:

- For each tree in the forest:
  - Randomly sample the training data with replacement (bootstrap sampling).
  - Randomly select a subset of features to consider for splitting at each node.
  - Train a decision tree on the sampled data using the selected features.
- For making predictions:
  - For each new instance, pass it through each tree in the forest to obtain the predicted class.
  - Aggregate the predictions from all trees using majority voting to determine the final predicted class.

Due to the randomness introduced in the training process, Random Forests are less prone to overfitting compared to individual decision trees and often achieve better generalization performance. [**?**]

3.7 Gradient Boosting

Gradient Boosting is an ensemble learning method that builds a series of weak learners in a sequential manner. Each weak learner is trained to correct the errors made by the previous learners. The final prediction is made by combining the predictions from all weak learners. [**?**]

Gradient Boosting Algorithm works using these following steps:

- Initialize the model with a constant value, typically the mean of the target variable.
- For each iteration $m = 1$ to M:
  - Compute the pseudo-residuals, which are the negative gradients of the loss function with respect to the current model's predictions.
  - Train a weak learner (e.g., decision tree, logistic regression) on the pseudo-residuals.
  - Compute the optimal step size (learning rate) for the weak learner.
  - Update the model by adding the weighted predictions of the weak learner to the current model.
- For making predictions:
  - For each new instance, pass it through all weak learners and sum their weighted predictions to obtain the final predicted value.

Since the problem we're working on is a binary classification task, we will use logistic loss as the loss function for Gradient Boosting. The logistic loss is defined as:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \text{ [?]}$$

where y_i is the true label, $\hat{y}_i$ is the predicted probability, and N is the number of instances. For the weak learners, we will use decision trees with a maximum depth of 5.

3.8 Hyperparameter Tuning

For each classification model, we will perform hyperparameter tuning using Grid Search combined with 10-Fold Cross Validation to find the optimal set of hyperparameters that yield the best performance on the validation set. The hyperparameters to be tuned for each model are as follows:

Grid Search works by exhaustively searching through a specified subset of hyperparameters for each model. For each combination of hyperparameters, the model is trained and evaluated using cross-validation. The combination that yields the best average performance across the folds is selected as the optimal set of hyperparameters. [**?**]

4 Experiment and Results

After performing the experiments using the described models and methods. We obtained the results as shown in Table **??**.

Table 3: Results of Classification Models (rounded to 4 decimal places)

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Random Forest	0.9037	0.9065	0.9037	0.9043
Gradient Boosting	0.8994	0.9015	0.8994	0.8999
Logistic Regression	0.8100	0.8696	0.8100	0.8271
LDA	0.7956	0.8676	0.7956	0.8155
K-Nearest Neighbors	0.7675	0.8233	0.7675	0.7863
Benchmark Model	0.4781	0.6889	0.4781	0.5375

Using Grid Search with 10-Fold Cross Validation, we found the optimal hyperparameters for each model is as follows:

- **Logistic Regression**: Regularization strength $C = 1.0$, Solver type using Liblinear, and Penalty type using Ridge Regularization (L2).
- **LDA**: Solver Type using SVD (Singular Value Decomposition)
- **K-Nearest Neighbors**: Number of neighbors $K = 3$, Distance metric using Manhattan distance, and Weighting type using the inverse of their distance.
- **Random Forest**: Number of trees $n\_estimators = 250$, Maximum depth of each tree $max\_depth = 15$, Minimum samples per leaf $min\_samples\_leaf = 1$, Minimum samples per split $min\_samples\_split = 2$, and Criterion using Entropy.
- **Gradient Boosting**: Number of estimators $n\_estimators = 100$, Learning rate = 0.1, Maximum Depth for each estimator = 10, and Subsample = 0.6

Based on the results, we can see that the more complex models such as Random Forest and Gradient Boosting outperformed the simpler models like Logistic Regression, LDA, and KNN.

Random Forest achieved an accuracy of 90.37%, indicating that it correctly classified a high percentage of instances in the dataset. The F1-Score of 90.43% also suggests that the model has a good balance between precision and recall, suggesting that it is effective in identifying both positive and negative instances. Gradient Boosting also performed comparably well, achieving an accuracy and F1-Score of 89.94% and 89.99% respectively, which is slightly lower than Random Forest but still significantly better than the other models.

If we compare the simpler models such as Logistic Regression, LDA, and KNN, we can see that they achieved lower accuracy and F1-scores. Logistic Regression achieved an accuracy of 81.00%, LDA achieved 79.56%, and KNN achieved 76.75%. This indicates that these models may not be as effective in capturing the complex relationships in the dataset compared to the ensemble methods.

Due to it's superior performance, for production deployment, we recommend using either the Random Forest or Gradient Boosting model for predicting whether an increase in bike stock is needed.

5 Conclusion

Based on the analysis and experiments conducted in this project, we can conclude that the Random Forest is the most effective model for predicting whether an increase in bike stock is needed in Washington D.C. The model achieved the highest accuracy, recall, precision, and F1-score among all the models evaluated.

Experiment

November 29, 2025

1 Appendix A

[1]:
```
import pandas as pd
import pycaret
import numpy as np
import copy
```

1.1 Read Raw Data

[2]:
```
df_raw = pd.read_csv('training_data_ht2025.csv')
```

[3]:
```
df_raw
```

[3]:
```
      hour_of_day  day_of_week  month  holiday  weekday  summertime  temp  \
0               5            5      1        0        0           0  -7.2
1              21            4      1        0        1           0  -1.3
2              21            3      8        0        1           1  26.9
3               1            6      1        0        0           0   3.1
4              17            0      3        0        1           0  11.7
…             …           …     …      …        …         …   …
1595            3            5      6        0        0           1  21.5
1596           14            0      6        0        1           1  23.2
1597           13            0      3        0        1           1  13.9
1598           14            5      3        0        0           1  11.7
1599           22            6      2        0        0           0   4.2

       dew  humidity  precip  snow  snowdepth  windspeed  cloudcover  \
0    -15.0     53.68   0.000     0        0.0       16.3        31.6
1    -12.8     40.97   0.000     0        0.0       23.9        85.7
2     21.8     73.39   0.000     0        0.0        0.0        81.1
3     -4.0     59.74   0.000     0        0.0       19.2         0.0
4    -11.4     18.71   0.000     0        0.0       10.5        44.6
…     …       …       …      …        …          …          …
1595  19.4     87.68   0.000     0        0.0       10.6        24.4
1596  20.1     82.43   2.217     0        0.0        9.8        92.1
1597  -2.2     32.93   0.000     0        2.0       18.2        79.3
1598  -9.3     22.09   0.000     0        0.0        5.8        24.4
```

```
1599    1.7      84.11    1.081      0         0.0        21.9         97.4

      visibility    increase_stock
0           16.0   low_bike_demand
1           16.0   low_bike_demand
2           16.0   low_bike_demand
3           16.0   low_bike_demand
4           16.0   low_bike_demand
…            …              …
1595        16.0   low_bike_demand
1596        10.4   low_bike_demand
1597        16.0   low_bike_demand
1598        16.0  high_bike_demand
1599        16.0   low_bike_demand

[1600 rows x 16 columns]
```

1.1.1 Check Data Contents

[4]:
```
df_raw['increase_stock'].value_counts()
```

[4]:
```
increase_stock
low_bike_demand     1312
high_bike_demand     288
Name: count, dtype: int64
```

Based on above code that shows the values on the target variable *increase_stock*, there's a problem of Imbalance Dataset, where the records with low bike demand are much more compared to records with high bike demands.

[5]:
```
df_raw['holiday'].value_counts()
```

[5]:
```
holiday
0    1547
1      53
Name: count, dtype: int64
```

[6]:
```
df_raw['weekday'].value_counts()
```

[6]:
```
weekday
1    1136
0     464
Name: count, dtype: int64
```

[7]:
```
df_raw['summertime'].value_counts()
```

[7]:
```
summertime
1    1030
```

```
0 236  570
Name: count, dtype: int64
```

[8]:
```
df_raw['increase_stock'].value_counts()
```

[8]:
```
increase_stock
low_bike_demand     1312
high_bike_demand     288
Name: count, dtype: int64
```

[9]:
```
df_raw['hour_of_day'].value_counts()
```

[9]:
```
hour_of_day
0     84
16    83
13    78
7     77
23    77
17    72
21    71
14    70
3     69
4     69
5     67
1     67
8     66
12    66
15    64
10    63
9     63
6     62
11    58
19    57
22    56
20    55
18    53
2     53
Name: count, dtype: int64
```

[10]:
```
df_raw['day_of_week'].value_counts()
```

[10]:
```
day_of_week
4    242
0    240
5    233
6    231
2    223
```

```
3 237 220
1     211
Name: count, dtype: int64
```

[11]:
```
df_raw['month'].value_counts()
```

[11]:
```
month
3     147
11    141
4     140
6     137
7     136
12    136
5     133
9     131
2     131
1     128
8     121
10    119
Name: count, dtype: int64
```

[12]:
```
df_raw['snow'].value_counts()
```

[12]:
```
snow
0    1600
Name: count, dtype: int64
```

Note: The Snow Features only have one value: 0, and therefore isn't really useful for this dataset. And In Feature selection we can drop them.

[13]:
```
df_raw.isnull().sum()
```

[13]:
```
hour_of_day       0
day_of_week       0
month             0
holiday           0
weekday           0
summertime        0
temp              0
dew               0
humidity          0
precip            0
snow              0
snowdepth         0
windspeed         0
cloudcover        0
visibility        0
increase_stock    0
```

```
dtype: int64
```

Based on the checking results, there're no NaN values, so we can proceed with feature processing.

Cyclical Encoding Function to encode Ordinal Variables into Cyclical encoding using Sine and Cosine

[14]:
```
def cyclical_encode(x, max_val, start_val=0):
    x_arr = np.asarray(x, dtype=float)

    # If values are 1..period (e.g. months 1..12), shift to 0..period-1
    if not start_val:
        x_arr = x_arr - 1

    angle = 2 * np.pi * x_arr / max_val
    sin_x = np.sin(angle)
    cos_x = np.cos(angle)

    if np.isscalar(x):
        return float(sin_x), float(cos_x)

    return sin_x, cos_x
```

[ ]:
```

```

[15]:
```
# Transform Cyclical Values for hour_of_day

# Transform Cyclical Values for hour_of_day
# Hour of day ranges from 0 to 23
df_hour_feat = pd.DataFrame(df_raw['hour_of_day'].apply(lambda x:␣
 ↪cyclical_encode(x, max(df_raw['hour_of_day']), min(df_raw['hour_of_day']))).
 ↪to_list())
df_hour_feat.columns = ['hour_of_day_sin', 'hour_of_day_cos']
df_hour_feat


# Transform Cyclical Values for day_of_week
# Day of week ranges from 0 (Monday) to 6 (Sunday)
df_day_feat = pd.DataFrame(df_raw['day_of_week'].apply(lambda x:␣
 ↪cyclical_encode(x, 6, 0)).to_list())
df_day_feat.columns = ['day_of_week_sin', 'day_of_week_cos']
df_day_feat

# Transform Cyclical Values for month
# Month ranges from 1 to 12
df_month_feat = pd.DataFrame(df_raw['month'].apply(lambda x: cyclical_encode(x,␣
 ↪12, 1)).to_list())
```

```
df_month_feat.columns = ['month_sin', 'month_cos']
df_month_feat

# Concat Cyclical Features
df_cyclical = pd.concat([df_hour_feat,df_day_feat, df_month_feat], axis=1)
df_cyclical
```

[15]:

```
      hour_of_day_sin  hour_of_day_cos  day_of_week_sin  day_of_week_cos  \
0            0.887885         0.460065    -8.660254e-01             -0.5
1           -0.730836         0.682553     1.224647e-16             -1.0
2           -0.730836         0.682553     8.660254e-01             -0.5
3            0.000000         1.000000    -8.660254e-01              0.5
4           -0.942261        -0.334880    -8.660254e-01              0.5
…                 …                …                …                …
1595         0.519584         0.854419    -8.660254e-01             -0.5
1596        -0.398401        -0.917211    -8.660254e-01              0.5
1597        -0.136167        -0.990686    -8.660254e-01              0.5
1598        -0.398401        -0.917211    -8.660254e-01             -0.5
1599        -0.519584         0.854419    -8.660254e-01              0.5

         month_sin     month_cos
0     5.000000e-01  8.660254e-01
1     5.000000e-01  8.660254e-01
2    -8.660254e-01 -5.000000e-01
3     5.000000e-01  8.660254e-01
4     1.000000e+00  6.123234e-17
…              …             …
1595  1.224647e-16 -1.000000e+00
1596  1.224647e-16 -1.000000e+00
1597  1.000000e+00  6.123234e-17
1598  1.000000e+00  6.123234e-17
1599  8.660254e-01  5.000000e-01

[1600 rows x 6 columns]
```

[16]:

```
# Creating Final Feature and Target DataFrames
df_features = copy.deepcopy(df_raw)

# Creating Target Variable and Mapping text to binary
df_target = df_features['increase_stock'].map({'low_bike_demand': 0,␣
 ↪'high_bike_demand': 1})

# Dropping original cyclical columns and target from features
df_features = df_features.drop(['hour_of_day', 'day_of_week', 'month',␣
 ↪'increase_stock'], axis=1)

# Concatenating Cyclical Features to Features DataFrame
```

```
df_features = pd.concat([df_cyclical, df_features], axis=1)

df_features
```

[16]:
```
      hour_of_day_sin  hour_of_day_cos  day_of_week_sin  day_of_week_cos  \
0            0.887885         0.460065    -8.660254e-01             -0.5
1           -0.730836         0.682553     1.224647e-16             -1.0
2           -0.730836         0.682553     8.660254e-01             -0.5
3            0.000000         1.000000    -8.660254e-01              0.5
4           -0.942261        -0.334880    -8.660254e-01              0.5
…                 …                …                …                …
1595         0.519584         0.854419    -8.660254e-01             -0.5
1596        -0.398401        -0.917211    -8.660254e-01              0.5
1597        -0.136167        -0.990686    -8.660254e-01              0.5
1598        -0.398401        -0.917211    -8.660254e-01             -0.5
1599        -0.519584         0.854419    -8.660254e-01              0.5

         month_sin     month_cos  holiday  weekday  summertime  temp   dew  \
0     5.000000e-01  8.660254e-01        0        0           0  -7.2 -15.0
1     5.000000e-01  8.660254e-01        0        1           0  -1.3 -12.8
2    -8.660254e-01 -5.000000e-01        0        1           1  26.9  21.8
3     5.000000e-01  8.660254e-01        0        0           0   3.1  -4.0
4     1.000000e+00  6.123234e-17        0        1           0  11.7 -11.4
…              …             …      …        …           …    …    …
1595  1.224647e-16 -1.000000e+00        0        0           1  21.5  19.4
1596  1.224647e-16 -1.000000e+00        0        1           1  23.2  20.1
1597  1.000000e+00  6.123234e-17        0        1           1  13.9  -2.2
1598  1.000000e+00  6.123234e-17        0        0           1  11.7  -9.3
1599  8.660254e-01  5.000000e-01        0        0           0   4.2   1.7

      humidity  precip  snow  snowdepth  windspeed  cloudcover  visibility
0        53.68   0.000     0        0.0       16.3        31.6        16.0
1        40.97   0.000     0        0.0       23.9        85.7        16.0
2        73.39   0.000     0        0.0        0.0        81.1        16.0
3        59.74   0.000     0        0.0       19.2         0.0        16.0
4        18.71   0.000     0        0.0       10.5        44.6        16.0
…          …     …     …     …          …         …          …
1595     87.68   0.000     0        0.0       10.6        24.4        16.0
1596     82.43   2.217     0        0.0        9.8        92.1        10.4
1597     32.93   0.000     0        2.0       18.2        79.3        16.0
1598     22.09   0.000     0        0.0        5.8        24.4        16.0
1599     84.11   1.081     0        0.0       21.9        97.4        16.0

[1600 rows x 18 columns]
```

[17]:
```
df_target
```

```
[17]: 0  241    0
      1         0
      2         0
      3         0
      4         0
               ..
      1595      0
      1596      0
      1597      0
      1598      1
      1599      0
      Name: increase_stock, Length: 1600, dtype: int64
```

```
[18]: # Correlation Analysis between Features and Target
      df_corr = pd.concat([df_features, df_target], axis=1).corr()
      df_corr['increase_stock'].sort_values(key=abs, ascending=False)
```

```
[18]: increase_stock      1.000000
      hour_of_day_cos    -0.339960
      temp                0.336981
      humidity           -0.308726
      hour_of_day_sin    -0.308121
      summertime          0.216052
      month_cos          -0.169059
      dew                 0.132663
      weekday            -0.116446
      visibility          0.113443
      windspeed           0.096011
      month_sin          -0.092078
      day_of_week_sin    -0.088152
      precip             -0.059304
      snowdepth          -0.047526
      cloudcover         -0.045534
      day_of_week_cos    -0.031473
      holiday            -0.004909
      snow                     NaN
      Name: increase_stock, dtype: float64
```

Based on above result, The variables hour_of_day_cos, temp, humidity, hour_of_day_sin, and summertime held the biggest correlation to the target variable. Because of this, we could check the trends more based on those attributes

```
[19]: from imblearn.over_sampling import SMOTE

      # Applying SMOTE to Balance the Dataset
      smote = SMOTE(sampling_strategy='minority')

      print("Before SMOTE:\n", df_target.value_counts())
```

242

```
print()

X_resampled, y_resampled = smote.fit_resample(df_features, df_target)

print("After SMOTE:\n", y_resampled.value_counts())

# Note: Don't use SMOTE outside training data to avoid data leakage, this block␣
 ↪is just for experimentation purposes. In practice, SMOTE should only be␣
 ↪applied to the training set within each cross-validation folds.
```

```
Before SMOTE:
 increase_stock
0    1312
1     288
Name: count, dtype: int64

After SMOTE:
 increase_stock
0    1312
1    1312
Name: count, dtype: int64
```

1.2 Model Experiment

[20]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, roc_auc_score, classification_report
from sklearn.model_selection import KFold, train_test_split
from sklearn.model_selection import GridSearchCV
```

K-Fold Classification Report

[21]:
```
## Function to do K-Fold Cross Validation and return the classification report
def kfold_classification_reports(clf, X, y, n_splits=10, shuffle=True,␣
 ↪random_state=42):
    kf = KFold(n_splits=n_splits, shuffle=shuffle, random_state=random_state)
    reports = []

    for train_index, test_index in kf.split(X):
        # Applying SMOTE to Balance the Dataset
        smote = SMOTE(sampling_strategy='minority')


        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
        clf.fit(X_resampled, y_resampled)
        y_pred = clf.predict(X_test)

        report = classification_report(y_test, y_pred, output_dict=True)
        reports += [report]

    detailed = []
    for i, rep in enumerate(reports):
        fold_result = {
            "fold": i + 1,
            "accuracy": rep["accuracy"],
            "precision": rep["weighted avg"]["precision"],
            "recall": rep["weighted avg"]["recall"],
            "f1": rep["weighted avg"]["f1-score"],
        }
        detailed.append(fold_result)

    # compute averaged (generalized) metrics
    accuracies = [d["accuracy"] for d in detailed]
    precisions = [d["precision"] for d in detailed]
    recalls = [d["recall"] for d in detailed]
    f1s = [d["f1"] for d in detailed]

    report_dict = {
        "general": {
            "accuracy_mean": float(np.mean(accuracies)),
            "accuracy_std": float(np.std(accuracies)),
            "precision_mean": float(np.mean(precisions)),
            "precision_std": float(np.std(precisions)),
            "recall_mean": float(np.mean(recalls)),
            "recall_std": float(np.std(recalls)),
            "f1_mean": float(np.mean(f1s)),
            "f1_std": float(np.std(f1s)),
        },
        "detailed": detailed
    }

    return report_dict
```

1.3 Benchmark Model

For the benchmark model, we use a naive model that predict each instance as the majority class in the training dataset. This will provide a baseline accuracy to compare the performance of more sophisticated models.

[22]:
```
from sklearn.dummy import DummyClassifier
```

```
clf_dummy = DummyClassifier(strategy="stratified")
```

[23]:
```
reports = kfold_classification_reports(clf_dummy, df_features, df_target)
reports['general']
```

[23]:
```
{'accuracy_mean': 0.47625,
 'accuracy_std': 0.030593095626301036,
 'precision_mean': 0.6952224635705411,
 'precision_std': 0.032953301949774735,
 'recall_mean': 0.47625,
 'recall_std': 0.030593095626301036,
 'f1_mean': 0.5351422005856692,
 'f1_std': 0.028627093758612O7}
```

1.4 Logistic Regression

For Logistic Regression, we will do hyperparameter tuning on Regularization Strength (C), Penalty type (*penalty*), and Solver type (*solver*)

[ ]:
```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()

#Grid Search to find the best hyperparameters for Logistic Regression
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'C': [0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear', 'lbfgs', 'saga', 'newton-cg']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_lg = grid_search.fit(df_features, df_target)
```

[249]:
```
clf_grid_lg.best_params_
```

[249]:
```
{'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
```

[250]:
```
clf_lg = LogisticRegression(**clf_grid_lg.best_params_)
```

[251]:
```
reports = kfold_classification_reports(clf_lg, df_features, df_target)
reports['general']
```

```
[251]: {'accuracy_mean': 0.8099999999999999,
        'accuracy_std': 0.026545950726994134,
        'precision_mean': 0.8695895287253285,
        'precision_std': 0.016029365215753214,
        'recall_mean': 0.8099999999999999,
        'recall_std': 0.026545950726994134,
        'f1_mean': 0.8271489924908391,
        'f1_std': 0.02043318475431806}
```

1.5 Linear Discriminant Analysis (LDA)

For LDA, we us Hyperparameter tuning for Solver Type (*solver*)

```
[252]: from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

       clf = LinearDiscriminantAnalysis()

       # Grid Search to find the best hyperparameters for LDA
       grid_search = GridSearchCV(
           estimator=clf,
           param_grid={
               'solver': ['svd', 'lsqr', 'eigen']
           },
           scoring='f1',
           cv=10,
           n_jobs=-1,
           verbose=1
       )
       clf_grid_lda = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 3 candidates, totalling 30 fits
```

```
[253]: clf_grid_lda.best_params_
```

```
[253]: {'solver': 'svd'}
```

```
[254]: clf_lda = LinearDiscriminantAnalysis(**clf_grid_lda.best_params_)
       clf_lda
```

```
[254]: LinearDiscriminantAnalysis()
```

```
[255]: reports = kfold_classification_reports(clf_lda, df_features, df_target)
       reports['general']
```

```
[255]: {'accuracy_mean': 0.795625,
        'accuracy_std': 0.0271065605527518,
        'precision_mean': 0.8675792283319778,
        'precision_std': 0.013589091958582378,
        'recall_mean': 0.795625,
        'recall_std': 0.0271065605527518,
        'f1_mean': 0.8155870764945796,
        'f1_std': 0.020417818352304083}
```

1.6 K Nearest Neighbor (KNN)

For K Nearest Neighbor, we use Hyperparameter tuning for selecting the numer of neighbors (*n_neighbors*), weight function used in prediction (*weights*), and the metric used for distance computation (*metric*)

[256]:
```python
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier()

#Grid Search to find the best hyperparameters for KNN
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_neighbors': range(3, 21, 2),
        'weights': ['uniform', 'distance'],
        'metric': ['euclidean', 'manhattan', 'minkowski']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_knn = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 54 candidates, totalling 540 fits
```

[257]:
```python
clf_grid_knn.best_params_
```

[257]:
```
{'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'distance'}
```

[258]:
```python
clf_knn = KNeighborsClassifier(**clf_grid_knn.best_params_)
clf_knn
```

[258]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=3, weights='distance')
```

[259]:
```python
reports = kfold_classification_reports(clf_knn, df_features, df_target)
reports['general']
```

[259]:
```
{'accuracy_mean': 0.7675,
 'accuracy_std': 0.020116846174288848,
 'precision_mean': 0.823321421215026,
 'precision_std': 0.026099143909350715,
 'recall_mean': 0.7675,
 'recall_std': 0.020116846174288848,
 'f1_mean': 0.7863382939494521,
 'f1_std': 0.020420617255737072}
```

1.7 Random Forest Classifier

For Random Forest Classifier, we use Hyperparameter tuning for selecting number of trees (*n_estimator*), maximum depth of each trees (*max_depth*), minimum number of samples required to split (*min_samples_split*), and minimum samples required to be a leaf node (*min_samples_leaf*)

[260]:
```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier()
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_estimators': [20, 50, 100, 200, 250],
        'max_depth': [3, 5, 8, 10, 15, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'criterion': ['gini', 'entropy']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_rf = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 540 candidates, totalling 5400 fits
```

[261]:
```
clf_grid_rf.best_params_
```

[261]:
```
{'criterion': 'entropy',
 'max_depth': 15,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 250}
```

[262]:
```
clf_rf = RandomForestClassifier(**clf_grid_rf.best_params_)
clf_rf
```

[262]:
```
RandomForestClassifier(criterion='entropy', max_depth=15, n_estimators=250)
```

[263]:
```
reports = kfold_classification_reports(clf_rf, df_features, df_target)
reports['general']
```

[263]:
```
{'accuracy_mean': 0.9037500000000002,
 'accuracy_std': 0.02186606960566987,
 'precision_mean': 0.9065025661469864,
 'precision_std': 0.019154309103939216,
 'recall_mean': 0.9037500000000002,
 'recall_std': 0.02186606960566987,
 'f1_mean': 0.9043197968362898,
 'f1_std': 0.02048728351994524}
```

1.8 Gradient Boosting

For Gradient Boosting, we use Hyperparameter tuning to determine, the number of sequential estimator (*n_estimators*), the learning rate (*learning_rate*), Maximum depth of the individual regression estimators (*max_depth*), and the fraction of samples to be used for fitting the individual estimators (*subsample*).

[264]:
```python
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier()
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.5],
        'max_depth': [3, 5, 8, 10],
        'subsample': [0.6, 0.8, 1.0]
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_gb = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 180 candidates, totalling 1800 fits
```

[267]:
```python
clf_grid_gb.best_params_
```

[267]:
```
{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 100, 'subsample': 0.6}
```

[268]:
```python
clf_gb = GradientBoostingClassifier(**clf_grid_gb.best_params_)
clf_gb
```

[268]:
```
GradientBoostingClassifier(max_depth=10, subsample=0.6)
```

[269]:
```python
reports = kfold_classification_reports(clf_gb, df_features, df_target)
reports['general']
```

[269]:
```
{'accuracy_mean': 0.899375,
 'accuracy_std': 0.02043472840533977,
 'precision_mean': 0.9014960071645403,
 'precision_std': 0.018078538997051353,
 'recall_mean': 0.899375,
 'recall_std': 0.02043472840533977,
 'f1_mean': 0.8998663647708327,
 'f1_std': 0.01972180961714667}
```

1.8.1 Saving Model

Saving all model so we can use it later

[270]:
```
# Applying SMOTE to Balance the Full Training Dataset
smote = SMOTE(sampling_strategy='minority')

print("Before SMOTE:\n", df_target.value_counts())

print()

X_resampled, y_resampled = smote.fit_resample(df_features, df_target)

print("After SMOTE:\n", y_resampled.value_counts())
```

```
Before SMOTE:
 increase_stock
0    1312
1     288
Name: count, dtype: int64

After SMOTE:
 increase_stock
0    1312
1    1312
Name: count, dtype: int64
```

[271]:
```
import pickle

# Save the Dummy Classifier model used as benchmark
with open('clf_dummy.pkl', 'wb') as f:
    pickle.dump(clf_dummy, f)
```

[272]:
```
# Train the Logistic Regression model on the full training dataset
clf_lg = LogisticRegression(**clf_grid_lg.best_params_)
clf_lg.fit(X_resampled, y_resampled)

# Save the trained Logistic Regression model
with open('clf_lg.pkl', 'wb') as f:
  250pickle.dump(clf_lg, f)
```

[ ]:
```
# Train the LDA model on the full training dataset
clf_lda = LinearDiscriminantAnalysis(**clf_grid_lda.best_params_)
clf_lda.fit(X_resampled, y_resampled)


# Save the trained LDA model
with open('clf_lda.pkl', 'wb') as f:
    pickle.dump(clf_lda, f)
```

[274]:
```
# Train the KNN model on the full training dataset
clf_knn = KNeighborsClassifier(**clf_grid_knn.best_params_)
clf_knn.fit(X_resampled, y_resampled)

# Save the trained KNN model
with open('clf_knn.pkl', 'wb') as f:
    pickle.dump(clf_knn, f)
```

[275]:
```
# Train the Random Forest Classifier model on the full training dataset
clf_rf = RandomForestClassifier(**clf_grid_rf.best_params_)
clf_rf.fit(X_resampled, y_resampled)

# Save the trained Random Forest Classifier model
with open('clf_rf.pkl', 'wb') as f:
    pickle.dump(clf_rf, f)
```

[276]:
```
# Train the Gradient Boosting Classifier model on the full training dataset
clf_gb = GradientBoostingClassifier(**clf_grid_gb.best_params_)
clf_gb.fit(X_resampled, y_resampled)

# Save the trained Gradient Boosting Classifier model
with open('clf_gb.pkl', 'wb') as f:
    pickle.dump(clf_gb, f)
```

[ ]:
```

```