Do we need more bikes?
Project in Statistical Machine Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

In this project, four statistical machine learning classification methods were trained and evaluated using a dataset containing hourly bike rental data from the District Department of Transportation in Washington, D.C. The dataset includes various temporal and meteorological features, such as time of day, weather conditions and temperature, to determine factors influencing bike demand. The classification models implemented included Logistic regression, Linear discriminant analysis (LDA), K-nearest neighbours (k-NN) and Random forest. The objective was to identify the model that most accurately predicts whether an increase in the number of bikes is necessary based on the given features. The models were assessed based on accuracy, precision, recall and F1-score. Hyperparameter tuning and cross-validation were performed to optimize performance. Among the tested models, Random forest demonstrated the best predictive capability by performing best on key evaluation metrics, balancing precision and recall effectively. The findings highlight the potential of machine learning models to support demand forecasting in bike-sharing systems. Number of group member: **4**

1 Introduction

Classification is a machine learning technique widely used in areas such as anomaly detection. It automatically categorises data into predefined classes based on various input features and can be applied to demand forecasting, such as predicting the need for additional bikes in a bike-sharing system. Ensuring sufficient bike availability is crucial as shortages could discourage users and lead to increased car usage and carbon emissions. The District Department of Transportation in Washington, D.C. wants to address this with a predictive model to determine when more bikes are needed. A dataset of 1600 observations will be preprocessed and used to train four classification models: Logistic regression, Linear discriminant analysis (LDA), K-nearest neighbours (k-NN) and Random forest. The best model will be selected based on precision, recall, F1-score and accuracy.

2 Data analysis

Prior to model development, an exploratory data analysis (EDA) was performed to gain insights into the data. The EDA was meant to answer questions on which features are numerical and categorical as well as examine greater trends between bike availability and time and weather features. See A for the data analyis code.

The categorical features are hour_of_day, day_of_week, month, holiday, weekday, summertime, and increase_stock. The rest are numerical.

The trends when comparing different hours, weeks, and months can be seen in Figure 1.

For the hours of day, the frequency to increase availability is highest between 15-19, while it is lowest between 23-06.

For the day of week, the need to increase availability is slightly higher during the weekend, especially on Saturdays.

For the months, the need to increase availability is lowest during winter months and peaks in April, June, September.

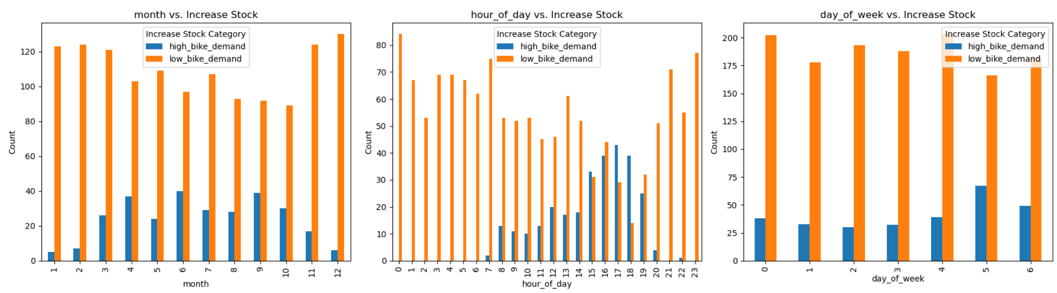

Figure 1: Time features plotted against `increase_stock`.

The relationships between the binary holidays and weekdays and the corresponding need to increase availability are displayed in Table 1 and Table 2, respectively. From Table 1 it seems that there there is no significant difference in availability need based on `holiday`. There does, however, seem to be higher bike demand when `weekday` is 0, i.e. it is the weekend, as seen in the difference in percentage in Table 2.

Table 1: Percentage distribution for Holidays

Holiday	Increase Stock	Percentage
0	Low Bike Demand	81.97%
	High Bike Demand	18.03%
1	Low Bike Demand	83.02%
	High Bike Demand	16.98%

Table 2: Percentage distribution for Weekdays

Weekday	Increase Stock	Percentage
0	Low Bike Demand	75.00%
	High Bike Demand	25.00%
1	Low Bike Demand	84.86%
	High Bike Demand	15.14%

The weather features trend against `increase_stock` can be seen in Figure 2.

For the temperature, the demand for bicycles is greater between 15-25 degrees Celsius. When the temperature is lower, between 8-22 degrees Celsius, the demand is generally lower.

Increased humidity, around 55-82, aligns with a lower demand, lower humidity, between 37-62, aligns with a higher demand.

The value distribution for windspeed, cloudcover, and visibility seem to be similar between high and low bike demand, indicating a lack of inference ability.

When the demand is high there is generally low or close to no precipitation. For low demand, there is generally moderate precipitation. However, there are days with no precipitation where the demand is still low. The same behavior can be seen for snowdepth.

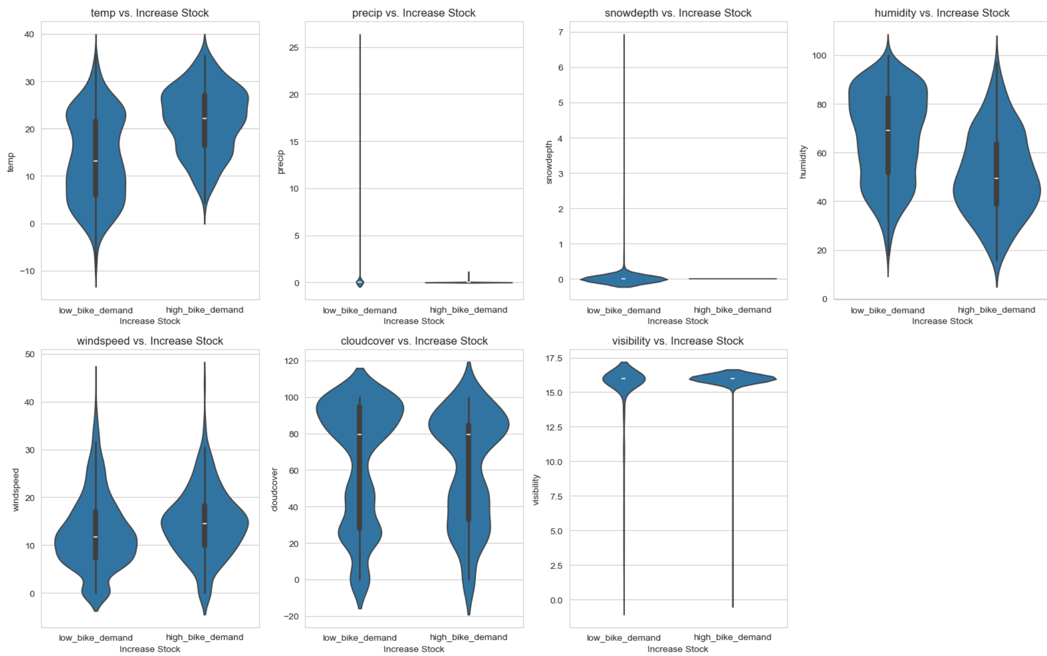

Figure 2: Weather features plotted against `increase_stock`.

2.1 Pre-processing

Following the EDA, a unified pre-processing was conducted, the code for which can be seen in Appendix B. Due to inconsistencies in the original feature, the pre-processing re-codes the feature `summertime` so that it is set to 1 if the month is between March and November. Furthermore, cyclical encoding of the time variables `day_of_week`, `hour_of_day`, and `month` was conducted through the function `cyclical_encoding` to catch the cyclical nature of the features. `increase_stock`'s values were converted to 1 if high_bike_demand and 0 if low_bike_demand. Then, the features `precip`, `snowdepth`, and `visibility` were converted to binary values where the values were set to 1 if the values were not their most common values which was 0, 0, and 16 respectively. Lastly, the features `holiday`, `snow` and the original time features were dropped. The pre-processing produced a new csv-file with data that was used for all models.

3 Model development

After the data had been analysed and processed the machine learning models were developed. Four models: logistic regression, linear discriminant analysis, K-nearest neighbours, and random forest and a naive classifier were developed.

First, all model developments used stratified 5-fold cross validation for training and validation because of the small dataset. Stratified cross validation was chosen because it keeps the proportion of classes for each fold which is useful for an imbalanced dataset such as the one used in the project. Further, to allow for hyperparameter tuning the data was initially split for all models into a training and test set. The training set was used in the cross validation and used for hyperparameter tuning while the test set was only used when evaluating the final model. The data was split into 80% training and 20% test data using random_state=42 between all models to guarantee consistency across models and reproducibility.

3.1 Evaluation metrics

To evaluate model performance accuracy, recall, precision and F1-score were used. F1-score was used as the cross validation metric. Due to the imbalanced dataset, accuracy was not single-handedly used to evaluate performance. Recall was seen as important due to the District Department of

Transportation's goal of identifying if bike availability should be increased. This makes it slightly more important to not miss cases when there actually is a need to increase availability. However, precision is also important to take into account since the department would not want to increase availability without there actually being demand for it. Thus, the F1-score, which averages precision and recall, is viewed as a useful metric when evaluating performance.

3.2 Naive classification model

A naive classifier was developed to bench mark the performance of the other models. The classifier was designed to only predict low_bike_demand and had an accuracy of roughly 0.82 due to the imbalanced dataset. See Appendix C for the naive classifier code.

3.3 Logistic regression

3.3.1 Mathematical description

Logistic Regression is a supervised binary classification problem that models the probability of an instance belonging to a particular class (Lindholm et al. [1]). The sigmoid function transforms the linear combination of input features into a probability bounded between 0 and 1.

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p = \theta^T \mathbf{x} \tag{1}$$

$$p(y = 1|\mathbf{x}) = g(\mathbf{x}) = \frac{e^{\theta^T \mathbf{x}}}{1 + e^{\theta^T \mathbf{x}}} \tag{2}$$

For binary classification, the decision is made based on a set threshold which is typically 0.5.

$$\hat{y} = \begin{cases} 1, & \text{if } g(\mathbf{x}) \geq \tau \\ 0, & \text{if } g(\mathbf{x}) < \tau \end{cases} \tag{3}$$

The model is trained by finding the optimal parameters θ that maximises the likelihood of correctly classifying the training data.

$$\ell(\theta) = \sum_{i=1}^{n} \left[y_i \log g(\mathbf{x}_i) + (1 - y_i) \log(1 - g(\mathbf{x}_i)) \right] \tag{4}$$

3.3.2 Method application

Logistic Regression was applied to the dataset after processing per subsection 2.1 and the common modeling choices. See Appendix D for the implementation code. Class imbalance was considered when splitting the data, with a balanced version tested, but the final model without balancing performed the best. The 18% minority class, while imbalanced, was perhaps not low enough (<10%) possibly explaining why balancing did not improve performance (van den Goorbergh et al. [3]). Grid search cross-validation, though computationally expensive, was preferred over random search to obtain the optimal hyperparameters (Sukamto et al. [2]). Regularisation strength (C) was tested at 0.1, 0.5, 1, 5, 10 and 20 along with liblinear and saga solvers and L1 (Lasso) and L2 (Ridge) penalties to prevent overfitting. The final model, optimised for F1-score to handle class imbalance, was trained with C = 20, solver = liblinear and penalty = L2. To further refine the classification decision, different threshold values were tested with 0.4 maximising the F1-score.

3.4 Linear discriminant analysis

3.4.1 Mathematical description

Linear discriminant analysis (LDA) can be expressed as Equation 5. This model follows from the generative model for classification and assuming a prior according to Equation 6, and that the conditional probability density of x for class m is a multivariate normal density with the mean described by Equation 7 and a covariance matrix, that is further assumed to be same between all classes, described by Equation 8.

$$\delta_m(x) = x^T \Sigma^{-1} \hat{\mu}_m - \frac{1}{2} \hat{\mu}_m^T \Sigma^{-1} \hat{\mu}_m + \log \hat{\pi}_m \tag{5}$$

Where:

$$\hat{\pi}_m = \frac{n_m}{n}, \quad m = 1, \ldots, M \tag{6}$$

$$\hat{\mu}_m = \frac{1}{n_m} \sum_{i:y_i=m} x_i, \quad m = 1, \ldots, M \tag{7}$$

$$\hat{\Sigma} = \frac{1}{n-M} \sum_{m=1}^{M} \sum_{i:y_i=m} (x_i - \hat{\mu}_m)(x_i - \hat{\mu}_m)^T \tag{8}$$

3.4.2 Method application

LDA was applied to the pre-processed data and used the common modelling choices, see Appendix E for the implementation code. Further, grid search was used to tune the LDA hyperparameters. Within each cross validation run the numerical weather and time features were scaled on the training set which was then used to transform the validation set. This approach was used to avoid data leakage from scaling on all training data before using cross validation. `humidity` and `cloudcover` were min-max normalized because of their defined ranges, while `temp`, `dew`, `windspeed`, and the time features were z-transform standardized. The lsqr solver and auto shrinkage parameters were found to be best from the hyperparameter tuning. Then, the classification threshold was tested by using the tuned model with different thresholds and choosing the best one. A threshold of 0.36 was found to perform best. Lastly, the final model was trained on all the training data and evaluated on the, until now, unseen test data.

3.5 K-nearest neighbours

3.5.1 Mathematical description

For a classification task, the k-NN method determines a new data point's class based on a majority vote of the k closest data points' classes (Lindholm et al., 2022). To calculate the distance between points, different distance metrics can be used to adapt the model to the characteristics of the input data. The Minkowski metric is a generalized expression which can simulate both Euclidean and Manhattan distance. The Minkowski distance, d between two points x_* and x_i with m input variables is given as:

$$d = \left(\sum_{j=1}^{m} |x_{ij} - x_{*j}|^p \right)^{\frac{1}{p}} \tag{9}$$

where (p = 1) gives the Manhattan distance, (p = 2) the Euclidean distance, and larger p-values render greater influence for greater distances in the input space. To avert overfitting and suppress sensitivity to noise, the method can aggregate output from multiple data points. However, using a large k might result in underfitting where patterns that characterise the different classes are lost.

3.5.2 Method application

An important aspect of using the k-NN method is to normalise all input variables to be of the similar scale (Lindholm et al. [1]). Utilising the preprocessed data, the input variables `humidity` and `cloudcover` were min-max-scaled since their upper and lower bounds were known. The variables `temp`, `dew`, and `windspeed` were scaled using Z-transform since they were assumed to have normally distributed values.

To evaluate the performance of different model variations the `GridSearchCV` function was used to test all combinations of different hyperparameters. The `distance metric` parameter was set to

Minkowski distance with another parameter p ranging between [1, 5], including Manhattan ($p = 1$) and Euclidean ($p = 2$) distance. The parameter `n_neighbours`, which determines the k-value, was set to range between [1, 50]. Additionally, the parameter `weights`, was added with the options `uniform`, which treats all neighbours' influence equally, and `distance`, which weighs neighbours' influence based on their distance to the new data point.

Manual examinations of different input variables were tested to achieve the greatest key metric scores, ultimately dropping `cloudcover`, `windspeed`, `dew`, and `snowdepth` from the input space. The function `GridSearchCV` then performed a 5-fold cross validation on the training set for all 490 combinations of hyperparameters resulting in 2450 fits. The best hyperparameters found were based on the averaged F1-score of each model fit's cross validation. Lastly, the model was evaluated on these parameters with the unseen test dataset to achieve a final performance report. See Appendix F for the implementation code for k-NN.

3.6 Random forest

3.6.1 Mathematical description

Tree-based methods in supervised machine learning builds on the concept of iteratively splitting data into smaller subsets based on specific decision rules. This is done by partitioning the input space into distinct regions, aiming to maximize class separation based on specific criterias (Lindholm et al. [1]). In this task the Gini impurity was used as the splitting criterion as described in Equation 10, where p_i is the proportion of class i instances in region R and C is the number of classes.

$$G(R) = 1 - \sum_{i=1}^{c} p_i^2 \tag{10}$$

In the formula, a lower $G(R)$ indicates higher purity, guiding the algorithm to split on the feature that minimizes impurity the most. By repeatedly splitting the data based on this guidance, tree-based models create clear decision rules that help classify new data points while also recognizing patterns and relationships between different features. However, the performance of (simple) classification and regression trees (CARTs) tend to overfit the training data, leading to high variance and reduced generalization performance. To address this, Random forest was chosen as the preferred tree-based method, as it combines bagging (bootstrap aggregation) and random feature selection when creating multiple decision trees and combining their predictions, reducing overfitting and increasing stability (Lindholm et al. [1]). Thus, this ensemble approach effectively reduces variance while maintaining low bias.

3.6.2 Method application

Random forest was applied to the pre-processed data and followed the common modeling choices. Given the imbalance in the dataset, where high bike demand was less frequent, class weighting was set to "balanced" when splitting the data to adjust for this.

Hyperparameter tuning was performed using grid search cross-validation over 16 parameter combinations, totaling 80 fits. The tested parameters included n_estimators (100, 300), max_depth (15, 20), min_samples_split (5, 10) and min_samples_leaf (2, 4), ensuring a balance between model complexity and generalization. max_features = "sqrt" was used to enhance model diversity by reducing feature correlation. The final model, optimized for F1-score (0.7018), was trained with n_estimators = 300, max_depth = 20, min_samples_split = 5, min_samples_leaf = 4 and class_weight = "balanced". See Appendix G for the implementation code for Random forest.

3.7 Performance evaluation

The naive classifier has an accuracy of roughly 0.82 when only predicting low_bike_demand. The models developed within this project all perform better than the naive classifier measured by accuracy, as seen in Figure 3. This indicates that the models have learned actual patterns from the training data and were able to generalize to unseen data better than a naive classifier. The highest accuracy was attained by the random forest model, which correctly classified 94% of the test data.

Furthermore, all models were better at classifying the class 0 (low_bike_demand), as seen by the higher F1-score. This can be explained by the unbalanced dataset making it more likely that the true value actually is 0 and there being more data corresponding to the classification 0 to train on. The F1-score was seen as a useful metric for this project and evaluating on the macro average F1-score could thus be fruitful. The metric averages the F1-score between the two classes equally, providing an unweighted estimate of the F1-score for both classes. It being unweighted is seen as favorable in this case due to class 1 prediction being slightly more important since the District Department of Transportation wanted to determine when to increase bike availability.

	precision	recall	f1-score	support
0	0.94	0.92	0.93	262
1	0.66	0.74	0.70	58
accuracy			0.88	320
macro avg	0.80	0.83	0.81	320
weighted avg	0.89	0.88	0.89	320

(a) Logistic regression classification report.

	precision	recall	f1-score	support
0	0.92	1.00	0.96	262
1	1.00	0.62	0.77	58
accuracy			0.93	320
macro avg	0.96	0.81	0.86	320
weighted avg	0.94	0.93	0.92	320

(b) Linear discriminant analysis classification report.

	precision	recall	f1-score	support
0	0.92	0.96	0.94	262
1	0.79	0.64	0.70	58
accuracy			0.90	320
macro avg	0.86	0.80	0.82	320
weighted avg	0.90	0.90	0.90	320

(c) K-nearest neighbour classification report.

	precision	recall	f1-score	support
0	0.97	0.96	0.96	262
1	0.82	0.84	0.83	58
accuracy			0.94	320
macro avg	0.89	0.90	0.90	320
weighted avg	0.94	0.94	0.94	320

(d) Random forest classification report.

Figure 3: Performance of all developed models shown through respective classification reports.

3.8 Model selection

The random forest model was chosen as the final model due to it outperforming every other model approach on all metrics. Due to the imbalance in the data, the occurrences of high bike demand become more important to capture. However, the bike rental service might not want to increase their stock on false occasions. Therefore, the F1-score proves as a significant evaluation metric since it provides an estimation of the model's ability to catch high bike demand instances, while avoiding false alarms. Since the F1-score is a balance between the precision and recall scores, imbalanced performance on these evaluation metrics can undermine the F1-score's actual significance. Therefore, due to the random forest model's balanced precision and recall scores, the F1-score proves the model's reliability to predict true instances of high bike demand while also maintaining very high performance in predicting low bike demand as well.

4 Conclusion

This project aimed to develop a classification model to predict whether an increase in bike availability would be necessary based on various temporal and weather-related features. Out of the four classification models: Logistic regression, Linear discriminant analysis, K-nearest neighbours and Random forest. The model best suited for this task was found to be Random forest, achieving a macro-average F1-score of 0.90. A key challenge during the project was dealing with the class imbalance, risking bias toward the majority class. To address this, different methods were tested, including class weighting, which improved Random forest but was less effective for Logistic regression. Future related work to this project could explore other classification methods such as boosting or using neural networks to offer potential further performance. To conclude, this project demonstrates that machine learning can enhance bike demand forecasting, while varying performances of the different classification models highlight the importance of careful model selection and hyper parameter tuning, tailored to the given task.

References

[1] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön. *Machine Learning A First Course for Engineers and Scientists*. Cambridge University Press, 2022.

[2] Sukamto, Hadiyanto, and Kurnianingsih. Knn optimization using grid search algorithm for preeclampsia imbalance class. *E3S Web of Conferences*, 448:2057—-2067, 2023. doi: https://doi.org/10.1051/e3sconf/202344802057.

[3] R. van den Goorbergh, M. van Smeden, D. Timmerman, and B. Van Calster. The harm of class imbalance corrections for risk prediction models: illustration and simulation using logistic regression. *Journal of the American Medical Informatics Association: JAMIA*, 29(9):1525—-1534, 2022. doi: https://doi.org/10.1093/jamia/ocac093.

A Appendix - Data analysis code

```python
# Import packages

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import sklearn.preprocessing as skl_pre
import sklearn.linear_model as skl_lm
import sklearn.discriminant_analysis as skl_da
import sklearn.neighbors as skl_nb
import sklearn.model_selection as skl_ms

#from IPython.display import set_matplotlib_formats
#set_matplotlib_formats('png')
from IPython.core.pylabtools import figsize
figsize(10, 6) # Width and hight
#plt.style.use('seaborn-white')

# Import training data
data = pd.read_csv('training_data_vt2025.csv')

print(data.head())

print(data.info())

print(data.describe().T)

# Plot distributions of each feature
data.hist(figsize=(12, 8), bins=30, edgecolor="black")
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()

# Plot boxplots for each feature
plt.figure(figsize=(12, 6))
data.plot(kind="box", subplots=True, layout=(5, 4), figsize=(14, 8), sharex=False, sharey=False)
plt.suptitle("Box Plots for Outlier Detection", fontsize=16)
plt.show()

# Plot correlation heatmap
df_encoded = data.copy()
df_encoded["increase_stock"] = df_encoded["increase_stock"].astype('category').cat.codes  # Encode target as numeric
corr = df_encoded.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.show()


# Plot time features against increase_stock

# Convert target variable to string if needed
data['increase_stock'] = data['increase_stock'].astype(str)

# Define categorical features
categorical_features = ['month', 'hour_of_day', 'day_of_week']

# Create a figure with subplots (1 row, 3 columns)
fig, axes = plt.subplots(1, 3, figsize=(18, 5))  # Adjust size as needed
```

```python

# Loop through features and plot each on a separate subplot
for ax, feature in zip(axes, categorical_features):
    data.groupby(feature)['increase_stock'].value_counts().unstack().plot(kind='bar', ax=ax)
    ax.set_title(f'{feature} vs. Increase Stock')
    ax.set_xlabel(feature)
    ax.set_ylabel('Count')
    ax.legend(title='Increase Stock Category')

# Adjust layout and display the plots
plt.tight_layout()
plt.show()


# Plot holiday and weekday against increase_stock

# Define categorical features
categorical_features = ['holiday', 'weekday']

# Create a figure with subplots (1 row, 3 columns)
fig, axes = plt.subplots(1, 2, figsize=(12, 4))  # Adjust size as needed

# Loop through features and plot each on a separate subplot
for ax, feature in zip(axes, categorical_features):
    data.groupby(feature)['increase_stock'].value_counts().unstack().plot(kind='bar', ax=ax)
    ax.set_title(f'{feature} vs. Increase Stock')
    ax.set_xlabel(feature)
    ax.set_ylabel('Count')
    ax.legend(title='Increase Stock Category')

# Adjust layout and display the plots
plt.tight_layout()
plt.show()


# Calculate percentage distribution of increase_stock for holidays
holiday_counts = data.groupby("holiday")["increase_stock"].value_counts(normalize=True) * 100

# Calculate percentage distribution of increase_stock for weekdays
weekday_counts = data.groupby("weekday")["increase_stock"].value_counts(normalize=True) * 100

# Display the results
print("Percentage Distribution of Increase Stock for Holidays:")
print(holiday_counts, "\n")

print("Percentage Distribution of Increase Stock for Weekdays:")
print(weekday_counts)


# Plot numerical features against increase_stock

# Define the numerical features to compare against increase_stock
numeric_features = ['temp', 'precip', 'snowdepth', 'humidity', 'windspeed', 'cloudcover', 'visibility']

# Set plot style
sns.set_style("whitegrid")

# Create subplots for better visualization
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(16, 10))
```

```
axes = axes.flatten()

# Plot each numeric feature against increase_stock
for i, feature in enumerate(numeric_features):
    if i < len(axes):  # Avoid index errors if fewer than 8 features
        sns.violinplot(x="increase_stock", y=feature, data=data, ax=
    axes[i])
        axes[i].set_title(f"{feature} vs. Increase Stock")
        axes[i].set_xlabel("Increase Stock")
        axes[i].set_ylabel(feature)

# Remove empty subplot (since we have 7 plots but a 2x4 grid)
fig.delaxes(axes[-1])

# Adjust layout
plt.tight_layout()
plt.show()
```

B Appendix - Pre-processing code

```python
# Import packages
import pandas as pd
import numpy as np

# Import data
data = pd.read_csv('training_data_vt2025.csv')

def cyclical_encoding(df, column, period):
    df[column + '_sin'] = np.round(np.sin(2 * np.pi * df[column] / period), 6)
    df[column + '_cos'] = np.round(np.cos(2 * np.pi * df[column] / period), 6)
    df.drop(columns=[column], inplace=True)  # Remove the original column
    return df

def pre_processing(data):
    # Make copy of dataset
    data_processed = data.copy()

    # Create new summertime feature
    data_processed['is_summer'] = ((data_processed['month'] >= 3) & (data_processed['month'] <= 11)).astype(int)

    # Normalize calendar data using cosine encoding
    data_processed = cyclical_encoding(data_processed, 'day_of_week', 7)
    data_processed = cyclical_encoding(data_processed, 'hour_of_day', 24)
    data_processed = cyclical_encoding(data_processed, 'month', 12)

    # Give target feature numerical values
    data_processed['increase_stock'] = data_processed['increase_stock'].replace({'high_bike_demand': 1, 'low_bike_demand': 0})

    # Create binary category of features
    data_processed['is_raining'] = (data_processed['precip'] != 0).astype(int)
    data_processed['is_snowing'] = (data_processed['snowdepth'] != 0).astype(int)
    data_processed['is_visible'] = (data_processed['visibility'] != 16).astype(int)

    # Drop columns
    data_processed = data_processed.drop(columns=['holiday', 'snow', 'snowdepth', 'precip', 'visibility', 'summertime'])

    return data_processed

new_data= pre_processing(data)

# Save the processed data
new_data.to_csv("preprocessed_data_1.csv", index=False)

print("Preprocessing complete. File saved as preprocessed_data_1.csv")
```

C Appendix - Naive classifier code

```
import pandas as pd
from sklearn.dummy import DummyClassifier
from pathlib import Path
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score
from sklearn.model_selection import train_test_split

# Read the preprocessed data

# Construct the full path to the CSV file
csv_file_path_pre_processed = Path.cwd().parent / 'preprocessed_data_1
    .csv'

# Read the CSV file using pandas
data = pd.read_csv(csv_file_path_pre_processed)

# Split the data into input values, X, and output value, y
X = data.drop(columns=['increase_stock'])
y = data['increase_stock']

# Split Data into Train & Test Sets (80% Train, 20% Test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, stratify=y, random_state=42)

# Create a Dummy Classifier (always predicts the majority class)
dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(X_train, y_train)

# Make predictions
y_dummy_pred = dummy_clf.predict(X_test)

# Evaluate performance
dummy_accuracy = accuracy_score(y_test, y_dummy_pred)
dummy_f1 = f1_score(y_test, y_dummy_pred, zero_division=1)  # Avoid
    division errors
dummy_roc_auc = roc_auc_score(y_test, y_dummy_pred)

# Print Results
print("Dummy Classifier Performance (Majority Class Strategy):")
print(f"Accuracy: {dummy_accuracy:.4f}")
print(f"F1-Score: {dummy_f1:.4f}")
print(f"ROC-AUC Score: {dummy_roc_auc:.4f}")
```

D Appendix - Logistic regression code

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold, train_test_split,
    GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, f1_score,
    roc_auc_score, accuracy_score, precision_score, recall_score

# Load Preprocessed data
file_path = "preprocessed_data_1.csv"
df = pd.read_csv(file_path)

# Drop increase stock
X = df.drop(columns=['increase_stock'])
y = df['increase_stock']

# Split data into train and test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Stratified k-fold
k_folds = 5
skf = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=42)

# Hyperparameter for gridsearch
param_grid = [
    {'C': [0.1, 0.5, 1, 5, 10, 20], 'penalty': ['l1'], 'solver': ['
    saga'], 'max_iter': [500]},
    {'C': [0.1, 0.5, 1, 5, 10, 20], 'penalty': ['l2'], 'solver': ['
    liblinear'],  'max_iter': [500]}
]

# Logistic regression model
logistic_model = LogisticRegression(random_state=42)

# Grid search cross-validation
grid_search = GridSearchCV(
    logistic_model, param_grid, cv=skf, scoring='f1', n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)

# Print the best parameters and f1 score
best_params = grid_search.best_params_
best_f1 = grid_search.best_score_

print("\n Best Hyperparameters Found:")
print(best_params)
print(f"Best F1 Score from Cross-Validation: {best_f1:.4f}")

# Train the final Model with best parameters
final_model = LogisticRegression(**best_params, random_state=42)
final_model.fit(X_train, y_train)

y_prob_test = final_model.predict_proba(X_test)[:, 1]

# Threshold tuning
thresholds = np.arange(0.4, 0.6, 0.8)
f1_scores = []

for threshold in thresholds:
```

```python
    y_pred = (y_prob_test >= threshold).astype(int)
    f1_scores.append(f1_score(y_test, y_pred))

best_threshold = thresholds[np.argmax(f1_scores)]
best_f1_final = max(f1_scores)

print(f"\n Best Threshold Found: {best_threshold:.2f}")
print(f"Best F1-Score at Best Threshold: {best_f1_final:.4f}")

# Best threshold
y_pred_final = (y_prob_test >= best_threshold).astype(int)

# Evaluate final model
final_accuracy = accuracy_score(y_test, y_pred_final)
final_precision = precision_score(y_test, y_pred_final)
final_recall = recall_score(y_test, y_pred_final)
final_f1 = f1_score(y_test, y_pred_final)
final_roc_auc = roc_auc_score(y_test, y_prob_test)

print("\n Final Model Evaluation:")
print(f"Accuracy: {final_accuracy:.4f}")
print(f"Precision: {final_precision:.4f}")
print(f"Recall: {final_recall:.4f}")
print(f"F1-Score: {final_f1:.4f}")
print(f"ROC-AUC Score: {final_roc_auc:.4f}")

# Classification report
print("\n Classification Report:\n")
print(classification_report(y_test, y_pred_final))
```

E Appendix - Linear discriminant analysis code

```python
import numpy as np
import pandas as pd
from pathlib import Path
from sklearn.model_selection import StratifiedKFold, GridSearchCV,
    train_test_split
from sklearn.metrics import classification_report, f1_score,
    roc_auc_score, accuracy_score, precision_score, recall_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
    RobustScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import confusion_matrix

# Read the preprocessed data

# Construct the full path to the CSV file
csv_file_path_pre_processed = Path.cwd().parent / 'preprocessed_data_1
    .csv'
csv_file_path_raw = Path.cwd().parent / 'training_data_vt2025.csv'

# Read the CSV file using pandas
pre_processed_data = pd.read_csv(csv_file_path_pre_processed)
raw_data = pd.read_csv(csv_file_path_raw)

# Split the data into input values, X, and output value, y
X = pre_processed_data.drop(columns=['increase_stock'])
y = pre_processed_data['increase_stock']

# Set aside test data that is kept unseen until final model evaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, stratify=y, random_state=42)

# Define column indices for different scalers
standardize_cols = ['temp','dew', 'windspeed','day_of_week_sin', '
    day_of_week_cos', 'hour_of_day_sin', 'hour_of_day_cos', 'month_sin
    ', 'month_cos']
normalize_cols = ['humidity', 'cloudcover']

# Define the column transformer with different scalers
preprocessor = ColumnTransformer([
    ('std_scaler', StandardScaler(), standardize_cols),
    ('minmax_scaler', MinMaxScaler(), normalize_cols)
], remainder='passthrough')

# Define Stratified K-Fold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Create a pipeline: Scaling + LDA
pipeline = Pipeline([
    ('preprocessor', preprocessor),  # Apply different scalers
    ('lda', LinearDiscriminantAnalysis())
])

param_grid = [
    {'lda__solver': ['svd']},  # SVD does not take shrinkage
    {'lda__solver': ['lsqr', 'eigen'], 'lda__shrinkage': ['auto', None
    ]}  # Only these solvers use shrinkage
]

# Perform Grid Search with Cross-Validation
grid_search = GridSearchCV(pipeline, param_grid, cv=skf, scoring='f1',
     n_jobs=-1, verbose=1)
```

```
grid_search.fit(X_train, y_train)

# Get the best parameters, but remove the 'lda__' prefix
best_params = {key.replace("lda__", ""): value for key, value in grid_search.best_params_.items()}

print("\n Best Hyperparameters Found:")
print(best_params)
best_f1 = grid_search.best_score_
print(f"Best F1 Score from Cross-Validation: {best_f1:.4f}")

final_lda_model = LinearDiscriminantAnalysis(**best_params)
final_lda_model.fit(X_train, y_train)

y_prob_test = final_lda_model.predict_proba(X_test)[:,1]

# Threshold tuning
thresholds = np.linspace(0,1,101)
f1_scores = []

for r in thresholds:
    y_pred = (y_prob_test >= r).astype(int)
    f1_scores.append(f1_score(y_test, y_pred))

best_threshold = thresholds[np.argmax(f1_scores)]
best_f1_final = max(f1_scores)

print(f"\n Best Threshold Found: {best_threshold:.2f}")
print(f"Best F1-Score at Best Threshold: {best_f1_final:.4f}")

# Best threshold
y_pred_final = (y_test-y_prob_test >= best_threshold).astype(int)

# Evaluate final model
final_accuracy = accuracy_score(y_test, y_pred_final)
final_precision = precision_score(y_test, y_pred_final)
final_recall = recall_score(y_test, y_pred_final)
final_f1 = f1_score(y_test, y_pred_final)
final_roc_auc = roc_auc_score(y_test, y_pred_final)

print("\n Final Model Evaluation:")
print(f"Accuracy: {final_accuracy:.4f}")
print(f"Precision: {final_precision:.4f}")
print(f"Recall: {final_recall:.4f}")
print(f"F1-Score: {final_f1:.4f}")
print(f"ROC-AUC Score: {final_roc_auc:.4f}")

# Classification report
print("\n Classification report:\n")
print(classification_report(y_test, y_pred_final))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred_final)

# Visualize cm using a heatmap
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Pred: 0', 'Pred: 1'], yticklabels=['True: 0', 'True: 1'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

F Appendix - K-nearest neighbor code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold, cross_val_predict, StratifiedKFold
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from collections import defaultdict
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score, precision_score, recall_score, roc_auc_score
from sklearn.preprocessing import LabelEncoder

# Load data
bikedata = pd.read_csv("training_data_vt2025.csv")


# Normalization scaler
n_scaler = MinMaxScaler()
bikedata["humidity"] = n_scaler.fit_transform(bikedata[["humidity"]])
bikedata["cloudcover"] = n_scaler.fit_transform(bikedata[["cloudcover"]])

# Standardization
s_scaler = StandardScaler()
bikedata["temp"] = s_scaler.fit_transform(bikedata[["temp"]])
bikedata["dew"] = s_scaler.fit_transform(bikedata[["dew"]])
bikedata["windspeed"] = s_scaler.fit_transform(bikedata[["windspeed"]])

# Binary transformation
bikedata['precip'] = bikedata['precip'].apply(lambda x: 1 if x > 0 else 0)
bikedata['snow'] = bikedata['snowdepth'].apply(lambda x: 1 if x > 0 else 0)
bikedata['visibility'] = bikedata['visibility'].apply(lambda x: 1 if x >= 16 else 0)

# Sine & Cosine encoding
bikedata['hour_sin'] = np.sin(2 * np.pi * bikedata['hour_of_day'] / 24)
bikedata['hour_cos'] = np.cos(2 * np.pi * bikedata['hour_of_day'] / 24)
bikedata['day_sin'] = np.sin(2 * np.pi * bikedata['day_of_week'] / 7)
bikedata['day_cos'] = np.cos(2 * np.pi * bikedata['day_of_week'] / 7)
bikedata['month_sin'] = np.sin(2 * np.pi * bikedata['month'] / 12)
bikedata['month_cos'] = np.cos(2 * np.pi * bikedata['month'] / 12)

#bikedata = pd.read_csv("fully_processed_data1.csv")

X = bikedata.drop(columns=['increase_stock', 'hour_of_day', 'day_of_week', 'month',  'snowdepth' , 'cloudcover', 'windspeed', 'dew'])
y = bikedata['increase_stock']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42)

# Encode the target variable
mapping = {'low_bike_demand': 0, 'high_bike_demand': 1}
y_train_encoded = [mapping[label] for label in y_train]
```

```
y_test_encoded = [mapping[label] for label in y_test]
'''
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
'''


kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
k_values = list(range(1, 50))

parameter_grid = {
    'n_neighbors' : (k_values),
    'weights' : ['uniform', 'distance'],
    'metric' : ['minkowski'],
    'p' : [1, 2, 3,4, 5]
}

KNN_model = KNeighborsClassifier()

grid_search = GridSearchCV(KNN_model, parameter_grid, cv = kf, scoring
     = 'f1', verbose = 1)

grid_search.fit(X_train, y_train_encoded)

best_parameters = grid_search.best_params_
best_f1 = grid_search.best_score_

print("\nBest Hyperparameters Found:")
print(best_parameters)
print("\n Best F1 Score from Cross-Validation:")
print(best_f1)



# Train final model with best hyperparameters
Final_KNN_model = KNeighborsClassifier(**best_parameters)
Final_KNN_model.fit(X_train, y_train_encoded)

y_pred = Final_KNN_model.predict(X_test)

# Evaluate model performance
print("Classification Report:\n")
print(classification_report(y_test_encoded, y_pred))
print("\n Confusion Matrix:\n")
print(confusion_matrix(y_test_encoded, y_pred))
```

G Appendix - Random forest code

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score, precision_score, recall_score, roc_auc_score

#  Load dataset
data = pd.read_csv("preprocessed_data_1.csv")

# Separate features (X) and target (y)
X = data.drop(columns=['increase_stock'])
y = data['increase_stock']

# Split data (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42)

# Decide amount of folds and do stratified K-fold
k_folds = 5
kfold = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=42) # shuffle ensures the data is randomized before splitting.

# Defines the hyperparameters to test in GridSearchCV
param_grid = {
  'n_estimators': [100, 300],      # Number of trees (more for stability)
  'max_depth': [15, 20],           # Maximum depth of each tree
  'min_samples_split': [5, 10],    # Minimum samples required to split a node (more to increase recall)
  'min_samples_leaf': [2, 4],      # Minimum samples per leaf node (Larger generelize better)
  'max_features': ['sqrt'],        # Each tree only uses square root of total features per split
  'class_weight': ['balanced']     # Adjusts weights to handle imbalanced data
}

# Initialize the tandom forest classifier
rf_model = RandomForestClassifier(random_state=42)

# Grid search with cross-validation to find the best hyperparameters
grid_search = GridSearchCV(
    estimator = rf_model,          # Random Forest as the base model
    param_grid = param_grid,       # Tests all combinations of hyperparameters
    cv = kfold,                    # Stratified K-Fold cross-validation (5 folds)
    scoring = 'f1',                # Optimizes the F1-score (useful for imbalanced datasets)
    n_jobs = -1,                   # Use all available CPU cores
    verbose = 1                    # Displays progress updates
)

# Train model with grid_search
grid_search.fit(X_train, y_train)

# Get best hyperparameters & best F1 score
best_params = grid_search.best_params_
best_f1 = grid_search.best_score_
```

```python
# Print best hyperparameters and F1 Score
print("\nBest Hyperparameters Found:")
print(best_params)
print("\nBest F1 Score from Cross-Validation:")
print(best_f1)

# Train final model with best hyperparameters
best_rf = RandomForestClassifier(**best_params, random_state=42)
best_rf.fit(X_train, y_train)

# Make predictions on test data
y_pred = best_rf.predict(X_test)

# Evaluate model performance
print("Classification Report:\n")
print(classification_report(y_test, y_pred)) # Confusion Matrix: Top: TP, FP & Bottom: FN, TN
print("\n Confusion Matrix:\n")
print(confusion_matrix(y_test, y_pred))

# Evaluating final model performance

f1 = f1_score(y_test, y_pred)                   # Measures the balance between precision & recall (good for imbalanced data)
accuracy = accuracy_score(y_test, y_pred)       # Overall correctness of predictions
precision = precision_score(y_test, y_pred)     # Proportion of positive predictions that were actually correct
recall = recall_score(y_test, y_pred)           # Proportion of actual positive cases correctly identified
roc_auc = roc_auc_score(y_test, y_pred)         # Measures models ability to distinguish between classes (higher = better)

# Print results
print("\nSummary of Model Performance:")
print(f"- F1 Score: {f1:.4f} ")
print(f"- Accuracy: {accuracy:.4f} ")
print(f"- Precision: {precision:.4f} ")
print(f"- Recall: {recall:.4f} ")
print(f"- AUC-ROC: {roc_auc:.4f} ")
```