Statistical Machine Learning
Uppsala University – Autumn 2024
Project Report

Lukas Bruna
lukas.bruna.9626@student.uu.se
Prakhar Dubey
prakhar.dubey.9915@student.uu.se
Sahil Mujahid
sahil.mujahid.1032@student.uu.se
Yashaswi Sood
yashaswi.sood.1789@student.uu.se

Abstract

This study aims to address the problem of optimizing bike availability in Washington, D.C.'s public bike-sharing system by predicting the need for additional bikes at specific hours. Using a data set of 1,600 hourly bike rental records, featuring variables such as date, time, weather conditions, and temperature, the task is formulated as a binary classification problem. The methodology involves data exploration and pre-processing, the application of multiple classification techniques, and performance evaluation. The outcome identifies the most effective model for deployment on unseen test data. Number of group members: 4.

1 Introduction

1.1 Problem Statement

Capital Bikeshare is a 24-hour public bicycle-sharing service in Washington, DC, providing transportation for thousands of residents and visitors daily. However, there are times when the demand for bikes exceeds the available supply, leading to service interruptions. Over time, this imbalance can drive people to use cars, contributing to higher CO2 emissions in the city. To address this issue, the District Department of Transportation aims to identify specific hours when increasing bike availability is necessary.
This project focuses on predicting the need for additional bikes using temporal and weather-related data. The resulting analysis can estimate the fluctuation in the demand for bikes to meet the needs of the public.

1.2 Feature Description

There are 1600 data points available in the given dataset. The following are the categorical and numerical features:

Categorical: hour_of_day, day_of_week, month, holiday, weekday, summertime
Numerical: temp, dew, humidity, precip, snow, snow_depth, windspeed, cloudcover, visibility
Label (Categorical): increase_stock

2 Exploratory Data Analysis

2.1 Feature Analysis

Before proceeding with model development, we conducted an exploratory data analysis to understand the relationships between input features and the target label. One key observation was the presence

of class imbalance in the target variable, with low demand being approximately four times more frequent than high demand. Temporal patterns also emerged as significant, with the hour of the day playing a crucial role in high demand. Demand peaks between 17:00 and 18:00 and drops to zero for a few hours after midnight. While the day of the week showed minimal impact, there was a notable increase in high demand during weekends, rising from 15.14% on weekdays to 25% on weekends. Similarly, seasonal trends were evident, as demand was low during winter months but increased during spring and autumn.

Holidays had a minimal effect on high demand, with an 18% occurrence on non-holidays compared to 17% on holidays. Weather conditions, however, exhibited a stronger influence. Rain negatively affected demand, with high demand dropping to 5.15% on rainy days compared to 19.38% on non-rainy days. Snow had an even more severe impact, with no high demand recorded on snowy days. These insights provide a comprehensive understanding of the factors influencing demand, forming a basis for model development.

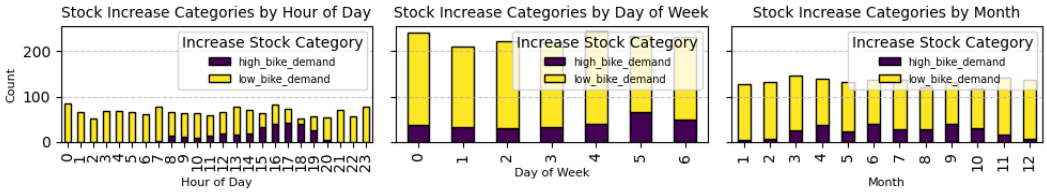

Figure 1: Dependence of Stock increase on hour, day and month

2.2 Data Split

The dataset consisted of a total of 1,600 rows, which were divided into three distinct subsets: training, validation, and test sets. The splitting was carried out following a 80-10-10% ratio, meaning 80% of the data (1,280 rows) was allocated to the training set, while the remaining 20% (160 rows each) was evenly split between validation and test sets. This approach ensured a sufficient volume of data for model training while reserving adequate samples for performance evaluation during both validation and testing phases. To maintain consistency and allow for reproducibility of the results, the splitting process was controlled by setting a random seed, with the value chosen as 42. This seed value guaranteed that the same subsets would be generated each time the split was performed, enabling consistent evaluation and comparison across different experiments or iterations of the model.

3 Model development

For the task of classification, the following methods were used and compared.

- Naive Classifier
- Logistic regression
- Discriminant Analysis (LDA, QDA)
- K-Nearest Neighbors
- Tree based methods: classification trees, random forest, bagging
- Boosting

In a machine learning workflow, starting with simple models like logistic regression offers a quick, interpretable baseline for linearly separable data. Discriminant analysis handles Gaussian class distributions, while k-nearest neighbors capture non-linear patterns but can be computationally expensive. Decision trees intuitively manage non-linear relationships but risk overfitting. More complex ensemble methods like random forests, bagging, and boosting enhance accuracy and manage intricate patterns effectively. Progressing from simple to complex models is essential for challenging datasets.

3.1 Naive Classifier

The first method implemented is a naive classifier which predicts all data points as *low_bike_demand*. Mathematically,

$$p(y = 1|\mathbf{x}) = 0$$

3.2 Logistic Regression

The principle of logistic regression lies in transforming z from linear regression into a [0,1] interval using a *logistic function* $h(z) = \frac{e^z}{1+e^z}$. This results in a [0,1] interval restricted logistic regression model:

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

This model contains unknown parameters θ we need to learn from the training data. We do this by minimizing logistic loss, learning the model thus amounts to solving Lindholm et al. [2022]:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ln\left(1 + e^{-y_i \theta^\top \mathbf{x}_i}\right)$$

Since this problem has no closed-form solution, we have to use numerical optimization through the python library. The "hard" prediction for an input $\mathbf{x}_*$ is then made with a threshold of $r = 0.5$, we explored other threshold values, however with our dataset we found no better option. Thus the $\hat{y}(\mathbf{x}_*)$ class is predicted to be the one with the highest probability.

$$\hat{y}(\mathbf{x}_*) = \begin{cases} 1 & \text{if } p(y = 1|\mathbf{x}) > r \\ -1 & \text{if } p(y = 1|\mathbf{x}) \leq r \end{cases}$$

3.3 Discriminant Analysis (LDA, QDA)

Discriminant Analysis is based on the Gaussian Mixture Model. GMM tries to model $p(x|y)$, as $p(y|x)$ can be derived from $p(x|y)$ by probability theory. GMMs primary assumption is that each $p(x|y)$ is a Gaussian distribution $p(x|y) = \mathcal{N}(x|\mu_y, \Sigma_y)$, where μ_y is the mean vector and Σ_y is the covariance matrix. This leaves us with $\theta = \{\mu_m, \Sigma_m, \pi_m\}_{m=1}^{M}$ unknown parameters we need to learn from the training data, M being the set of categorical y values and π_m their corresponding probability.
To make predictions from a generative GMM we compute the conditional distribution by applying said theory:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|y)}{\Sigma_{j=1}^{M} p(\mathbf{x}|y = j)} \rightarrow p(y = m|\mathbf{x}_*) = \frac{\hat{\pi}_m \mathcal{N}(\mathbf{x}_*|\hat{\mu}_m, \hat{\Sigma}_m)}{\Sigma_{j=1}^{M} \hat{\pi}_j \mathcal{N}(\mathbf{x}_*|\hat{\mu}_j, \hat{\Sigma}_j)} \tag{1}$$

Like with logistic regression, a "hard" prediction is then obtained as follows:

$$\hat{y}_* = \arg\max_{m} p(y = m|\mathbf{x}). \tag{2}$$

To obtain the Quadratic Discriminant Analysis classifier from GMM, we have to realize that "the logarithm of the Gaussian probability density function is a quadratic function in x" Lindholm et al. [2022] this means the classifiers decision boundary is also quadratic if we take the logarithm of 2 (which does not change the maximizing argument):

$$\hat{y}_* = \arg\max_{m} \left\{ \ln \hat{\pi}_m + \ln \mathcal{N}\left(\mathbf{x}_* \mid \hat{\mu}_m, \hat{\Sigma}_m\right) \right\}. \tag{3}$$

giving us the QDA classifier, furthermore by making additional simplifying assumptions, we can also obtain a special case of QDA, the Linear Discriminant Analysis classifier. Assuming that the covariance matrix in 1 is equal for all classes leads to a cancellation of all quadratic terms when computing the class predictions in 3. This means we only have one covariance matrix to learn and the decision boundary of this classifier is linear, the method is called LDA.

3.4 K-Nearest Neighbors

The k-Nearest neighbor algorithm is non parametric and instance based. It classifies a new observation based on the majority class of its k nearest neighbors in a feature space. The distance between a new observant and an existing observant is key and is calculated using Euclidean distance as the distance metric.

$$d(x, x_1) = \sqrt{\sum_{j=1}^{n} (x_j - x_{ij})^2}$$

where x_j and x_{ij} are the values of the j-th feature for the new observation and the i-th training observation, respectively. The training observations with the k smallest distances to x are the k nearest neighbors. For a classification task, the new observation is assigned to the majority class among the k nearest neighbors.

$$\hat{y} = \arg\max_{c} \sum_{i \in neigbors} 1(y_i = c)$$

Here 1 is an indicator function, and c iterates over all classes.

3.5 Tree-Based Methods

Tree-based methods are widely used for predictive modeling tasks. They are intuitive, interpretable, and can capture non-linear relationships effectively.

Classification and Regression Trees (CART) split data into subsets based on feature values to minimize impurity (for classification) or variance (for regression). The tree recursively partitions the feature space into regions associated with different predictions.

Decision Trees are the foundation of tree-based methods. They use cost functions to measure impurity or error in the splits. Common cost functions for classification include:

Gini Impurity:

$$G = 1 - \sum_{i=1}^{C} p_i^2$$

where p_i is the proportion of class i in the node.

Entropy:

$$H = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

Random Forests are an ensemble method that combines predictions from multiple decision trees. Each tree is trained on a random subset of the data and features, introducing diversity to reduce overfitting. The final prediction aggregates outputs from all trees.

Classification Prediction (Majority Voting):

$$\hat{y} = \arg\max_{k} \left(\sum_{t=1}^{T} I(h_t(x) = k) \right)$$

where $h_t(x)$ is the prediction of the t-th tree, and T is the total number of trees.

Bagging (Bootstrap Aggregating) trains multiple decision trees on different bootstrap samples of the dataset. This approach reduces variance by averaging predictions (for regression) or taking a majority vote (for classification).

Bootstrap Sampling: Each tree is trained on D_b, a bootstrap sample of the dataset D.

Classification Prediction (Voting):

$$\hat{y} = \arg\max_{k} \left(\sum_{b=1}^{B} I(h_b(x) = k) \right)$$

where B is the total number of bootstrap samples.

Tree-based methods like random forests and bagging extend the simple decision tree model to provide more robust and generalized predictions, making them highly effective for a variety of tasks.

3.6 Boosting

Boosting is an ensemble learning technique that combines predictions from multiple weak learners to create a strong model. It iteratively corrects the errors of previous models by assigning higher weights to misclassified data points. The weighted error at each iteration t is given below, where w_i are the weights and $1(\cdot)$ is the indicator function.

$$L_t = \sum_{i=1}^{n} w_i \cdot 1(y_i \neq h_t(x_i)),$$

The final model is the weighted sum of weak learners is given below, where α_t represents the weight of the t-th learner.:

$$F(x) = f_0(x) + \sum_{t=1}^{T} \alpha_t h_t(x),$$

In **AdaBoost**, the weight α_t of each weak learner depends on its classification error L_t:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - L_t}{L_t} \right).$$

Weights of data points are updated using the equation given below, and normalized to maintain a probability distribution. This ensures focus on harder-to-classify samples.

$$w_i^{(t+1)} = w_i^{(t)} \exp\left(\alpha_t \cdot 1(y_i \neq h_t(x_i))\right),$$

XGBoost enhances boosting by incorporating gradient-based optimization. At each step, it minimizes a second-order approximation of a loss function $L(F(x))$:

$$h_t(x) = \arg\min_h \sum_{i=1}^{n} \left[g_i h(x_i) + \frac{1}{2} h^2(x_i) h_i \right] + \Omega(h),$$

where g_i is the Gradient, h_i is the Hessian, and $\Omega(h)$ is the regularization term:

4 Results

The models were evaluated on the aforementioned test set for the sake of comparision and picking the supreme model.

4.1 Naive Classifier

While the test set accuracy of predicting all 0 values was 0.865, the macro F1 score was quite poor, with a value of 0.45, implying that the model didn't perform well on both classes.

4.2 Logistical Regression

Using the grid search cross-validation technique to perform an exhaustive search over parameters *penalty*, *C*, *solver*, and *max_iter* results in 600 candidates, fitting 5 folds for each leads to 3000 fits. Among these, *penalty* specifies the type of regularization, with *l1* enforcing sparsity by reducing less important feature weights to zero; *C* controls the trade-off between error minimization on the

training set and model simplicity, with lower values applying stronger regularization; *max_iter* sets the maximum number of iterations for convergence; and *solver* determines the optimization algorithm, with *liblinear* being efficient for smaller datasets. The best-found parameters were: *C*: 0.2336, *max_iter*: 3000, *penalty*: l1, *solver*: liblinear.

4.3 Discriminant Analysis (LDA, QDA)

The grid search cross-validation identified the best parameters for LDA as *shrinkage* = auto, *solver* = lsqr, and *store_covariance* = True. The *shrinkage* parameter regularizes the covariance matrix, with *auto* adaptively determining the shrinkage intensity to handle highly correlated data and reduce variance. The *solver* specifies the algorithm used for computations; *lsqr* is efficient for large datasets with linear discriminants. The *store_covariance* option, set to True, retains the computed covariance matrix for further analysis. For QDA, where invertible covariance matrices are critical, PCA is used to create linearly uncorrelated features, mitigating multicollinearity and ensuring numerical stability Næs and Mevik [2001].

4.4 K- Nearest Neighbors Algorithm

Hyperparameter tuning over *n_neighbors* and *weights* in a k-Nearest Neighbors (k-NN) model revealed the best configuration as $k = 11$ with *weights* = uniform. The *n_neighbors* parameter specifies how many nearest neighbors vote to determine the classification, with larger values smoothing decision boundaries to reduce variance. The *weights* parameter defines how neighbor distances influence voting: *uniform* gives equal weight to all neighbors, while *distance* assigns higher weight to closer neighbors. The uniform weighting balances training and test accuracy without overfitting. In contrast, *distance* weighting achieves 100% training accuracy, indicating excessive tailoring to the training set and reducing generalization reliability, despite a marginal test accuracy gain.

4.5 Trees

4.5.1 Decision Tree

The simplest tree method upon doing hyperparameter tuning via grid search gave reasonable training accuracy of 0.897 but not as good on test set with 0.862.

The best set of hyperparameters was as follows: 'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 10
Entropy works better for overlapping features like in this case, since it prefers splits that maximize homogenity in child nodes. The remaining hyperparameters are ideal to ensure a low variance, but balanced model.

4.5.2 Random Forest

Random forest performs pretty well on training data with an accuracy of 0.906 but not as well on test with 0.862 after hyperparameter optimization via grid search.

Set of hyperparameters chosen: Best Parameters: 'bootstrap': False, 'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 10, 'n_estimators': 10
Since the dataset is small, bootstrapping does not help as it cuts down the data even further. The rest of the hyperparameters ensure that the ensemble of trees is balanced while preventing overfitting.

4.5.3 Bagging

Bagging performed similar to Random Forest on training set with an accuracy of 0.903 but a little better on test with 0.868.

Hyperparameters after grid search: 'bootstrap': True, 'max_features': 1.0, 'max_samples': 1.0, 'n_estimators': 50, 'n_jobs': -1 bootstrap is set to true since bagging primarily benefits by bootstrapping. Number of estimators are higher to ensure a varied number of sample datasets are captured.

4.6 Boosting

4.6.1 AdaBoost

An *AdaBoostClassifier* with decision trees was optimized using grid search. The best hyperparameters were found as *n_estimators* = 50, controlling the number of weak learners combined to form the final model, and *learning_rate* = 0.9, which scales the contribution of each weak learner, balancing bias and variance. Using a coarse-to-fine grid search approach, the model achieved accuracies of 0.913 on training, 0.888 on validation, and 0.856 on the test set, demonstrating robust generalization.

4.6.2 XGBoost

An XGBoost model initially used default parameters, yielding a training accuracy of 1.00 and validation accuracy of 0.806, indicating high variance. To reduce overfitting, a grid search refined key hyperparameters: *n_estimators* = 50, controlling the number of boosting rounds; *learning_rate* = 0.1, determining the weight of each added tree; *gamma* = 1, setting the minimum loss reduction for further splits; and *reg_lambda* = 10, applying L2 regularization to limit model complexity. The tuned model achieved accuracies of 0.929 on training, 0.900 on validation, and 0.875 on the test set, striking a balance between bias and variance.

4.7 Model Comparison

The results are summarised in the table below:

Table 1: Model Performance Comparison

Model Type	**Train Accuracy**	**Val Accuracy**	**Test Accuracy**	Macro Avg. F1 Score
Naive Classifier	0.81	0.83	0.865	0.45
Logistic Regression	0.852	0.831	0.838	0.75
Discriminant Analysis LDA	0.852	0.825	0.831	0.75
Discriminant Analysis QDA	0.812	0.825	0.812	0.75
K-Nearest Neighbors (k=11, Uniform)	0.876	0.862	0.861	0.72
K-Nearest Neighbors (k=11, Distance)	1.000	0.875	0.876	0.74
Decision Tree	0.897	0.875	0.862	0.75
Random Forest	0.906	0.868	0.862	0.74
Bagging	0.903	0.868	0.868	0.76
AdaBoost	0.913	0.888	0.856	0.71
XGBoost	0.929	0.900	0.875	0.76

4.8 Inputs and Model Selection

4.8.1 Input Selection

For input selection, most of the variables in the dataset were considered, except for 'summertime' and 'snow.' The 'summertime' feature was excluded because it was redundant, given that the 'month' feature provided more granular information, while 'snow' was excluded as it consistently had a value of zero throughout the dataset. Various custom features, all boolean, were explored within the Decision Tree Classifier. These included *day*, which captured time-based information for Washington D.C.; *good_weather_humid_temp*, indicating favorable weather based on deviations from mean humidity and temperature; and *good_weather_rain_snow*, representing clear weather based on the absence of rain or snow. However, these custom features did not significantly enhance model performance, suggesting that more sophisticated numeric features might yield better results. Additionally, we experimented with a subset of top features identified through feature importance rankings, which produced results similar to the original model. The top features selected included ['hour_of_day', 'temp', 'humidity', 'windspeed', 'dew', 'cloudcover', 'month', 'day_of_week', 'precip', 'visibility', 'holiday'].

4.8.2 Model Selection

For model selection, we evaluated several algorithms. Logistic Regression, Linear Discriminant Analysis (LDA), and Quadratic Discriminant Analysis (QDA) demonstrated limited performance compared to other models. Among tree-based models, all three variations performed similarly, with the Bagging classifier showing a slight edge. Within boosted tree algorithms, XGBoost consistently outperformed other models in test set accuracy and F1 score, except for an overfitted K-Nearest Neighbors (KNN) model. In conclusion, **XGBoost** was selected as the final model for production due to its superior performance and balanced evaluation metrics.

5 Conclusion

In this project, we tackled the challenge of analyzing a small dataset to predict bike demand based on various features. Our exploration began with an in-depth analysis of the dataset, including the correlations between features and the target variable. To ensure a solid foundation for our work, we also discussed the mathematical principles underpinning the models, giving us a clear understanding of their strengths and limitations.

We implemented a range of models, including Logistic Regression, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), K-Nearest Neighbors (KNN), Decision Trees, Random Forest, Bagging, and Boosting techniques. In addition, we experimented with creating custom features and selecting feature subsets to improve model performance.

Among all the models tested, XGBoost emerged as the best-performing model, narrowly surpassing the alternatives in terms of test accuracy. Its robust performance makes it the best choice to *put into production*.

References

Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B. Schön. *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press, 2022. URL `https://smlbook.org`.

Tormod Næs and Bjørn-Helge Mevik. Understanding the collinearity problem in regression and discriminant analysis. *Journal of Chemometrics*, 15(4):413–426, 2001. doi: https://doi.org/10.1002/cem.676. URL `https://analyticalsciencejournals.onlinelibrary.wiley.com/doi/abs/10.1002/cem.676`.

A Appendix

A.1 Plots

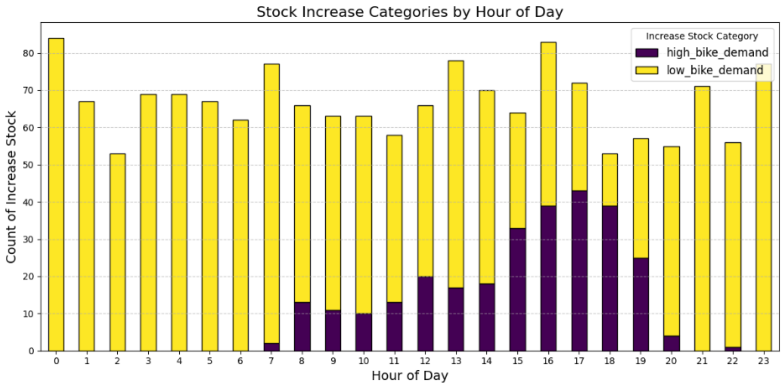

Figure 2: Dependence of Stock increase to the hour of the day

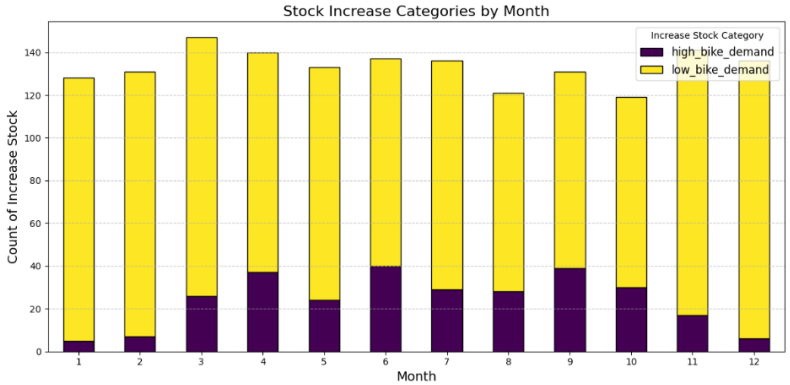

Figure 3: Dependence of Stock increase to the month

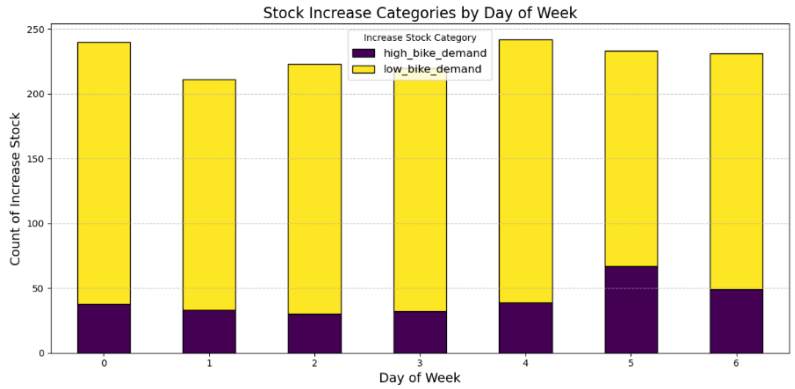

Figure 4: Dependence of Stock increase to the day of the week

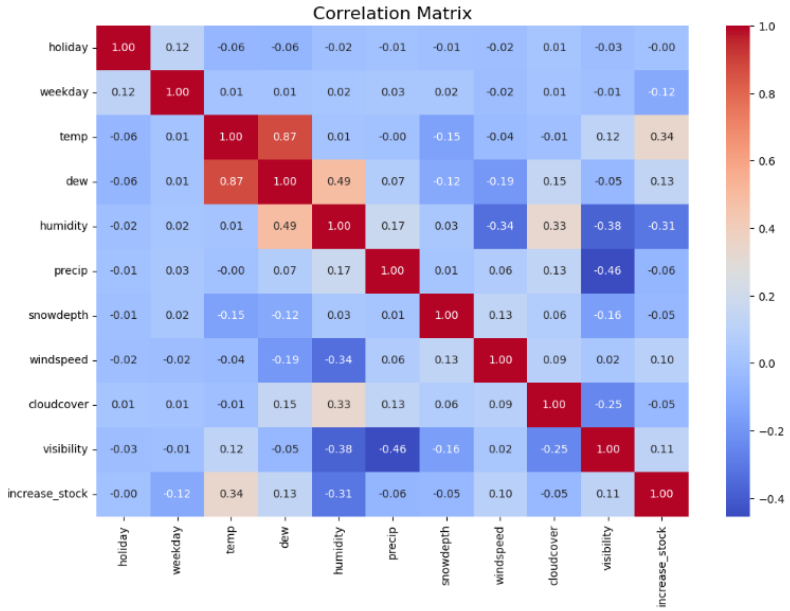

Figure 5: Correlation Matrix

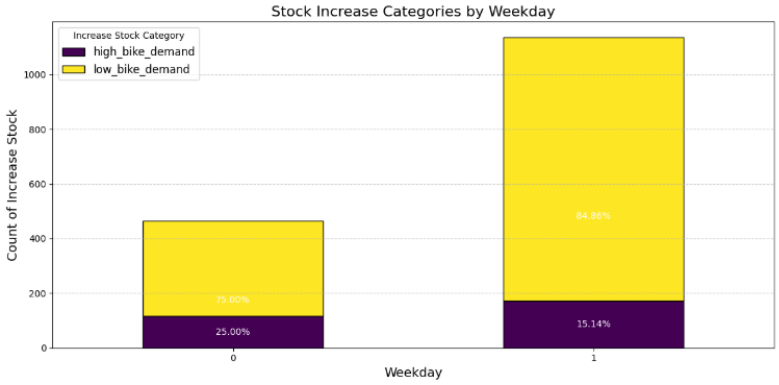

Figure 6: Dependence of Stock increase on weekdays

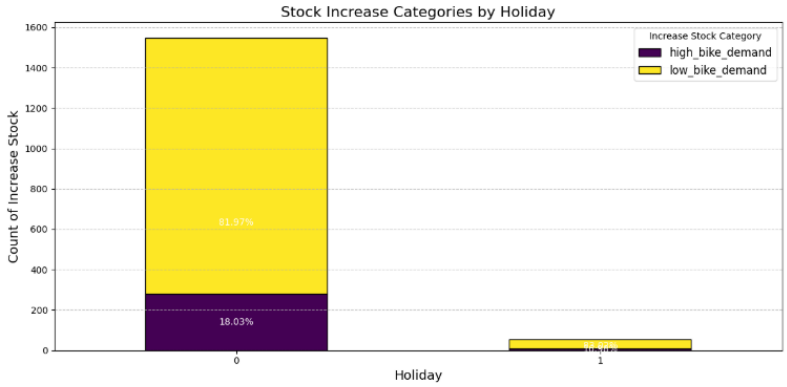

Figure 7: Dependence of Stock increase on Holidays

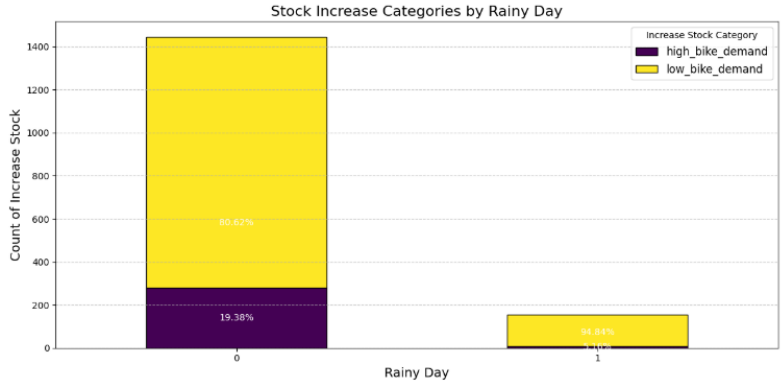

Figure 8: Dependence of Stock increase on Rainy days

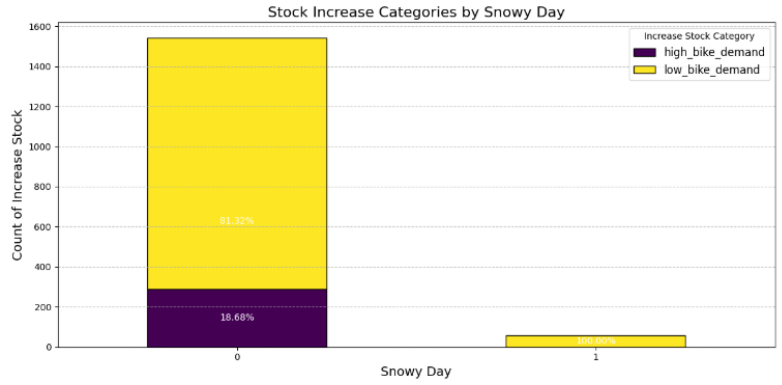

Figure 9: Dependence of Stock increase on Snowy days

A.2 Code

```
# -*- coding: utf-8 -*-
"""naive_classifier.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1lf5hhCLoxzOHsh6XAwqErLMk_vEWC3yv
```

```
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math as mt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.metrics import classification_report, accuracy_score, f1_score


from google.colab import drive
drive.mount('/content/gdrive')


data = pd.read_csv('/content/gdrive/MyDrive/SML_project/training_data_fall2024.csv')

data['increase_stock'] = data['increase_stock'].replace({
    'low_bike_demand': 0,
    'high_bike_demand': 1
    })

X = data.drop(columns=['increase_stock'])
y = data['increase_stock']

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1, random_state=
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.111, r

# Predict all test values as 0/low_bike_demand
y_pred = pd.Series([0]*y_test.shape[0])

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Accuracy Score:", accuracy_score(y_test, y_pred))

y_pred = pd.Series([0]*y_test.shape[0])

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Accuracy Score:", accuracy_score(y_test, y_pred))

# -*- coding: utf-8 -*-
"""LR_LDA_QDA.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1ibXm4k3Qquw-Tr3GUw_a5_ySPEs9YDJF
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegressionCV
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.metrics import precision_recall_curve, f1_score
from sklearn.metrics import roc_curve, roc_auc_score
import math as mt
```

```
#data import and cleaning
data = pd.read_csv("training_data_fall2024.csv")
data=data.drop(['snow', 'summertime'], axis = 1)

#80:10:10 split
X = data[['hour_of_day','day_of_week','month'
          ,'holiday','weekday','temp','dew',
          'humidity','precip','snowdepth',
          'windspeed','cloudcover','visibility']]
Y = data['increase_stock']
X_temp, X_test, Y_temp, Y_test = train_test_split(X, Y,
                                                  test_size=0.1,
                                                  random_state=42)
X_train, X_valid, Y_train, Y_valid = train_test_split(X_temp, Y_temp,
                                                      test_size=0.111,
                                                      random_state=42)

#standardization (leads to higher accuracies accross the board)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.fit_transform(X_valid)
X_test = scaler.fit_transform(X_test)

treshold = 0.5

#logistic regression model hyperparameter grid search tuning
#errors in output caused by incompatible solvers and penalties
from sklearn.model_selection import GridSearchCV

param_grid = {
    'penalty':['l1','l2','elasticnet'],
    'C' : np.logspace(-4,4,20),
    'solver': ['lbfgs','newton-cg','liblinear','sag','saga'],
    'max_iter'  : [3000,5000]
}

grid_search = GridSearchCV(
    estimator=LogisticRegression(random_state=42),
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train, Y_train)
Y_pred = grid_search.predict(X_valid)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

#logistic regression model
#liblinear came out to be the best solver as expected
#since it is a small dataset
#max_iter is set to 3000 to avoid the ConvergenceWarning
model = LogisticRegression(random_state=42, max_iter=3000,
                           penalty='l1', solver='liblinear',
                           C=0.23357214690901212)
model.fit(X_train,Y_train)
print(model)

train_predict = model.predict_proba(X_train)
prediction = np.empty(len(X_train),dtype=object)
prediction = np.where(train_predict[:,0]>=treshold,
                      'high_bike_demand','low_bike_demand')
print("Train Accuracy:",f"{np.mean(prediction == Y_train):.3f}")
```

```
test_predict = model.predict_proba(X_test)
prediction = np.empty(len(X_test),dtype=object)
prediction = np.where(test_predict[:,0]>=treshold,
                      'high_bike_demand','low_bike_demand')
print(pd.crosstab(prediction,Y_test))
# print(f"{f1_scores[best_index]:.2f}")
print("Test Accuracy:",f"{np.mean(prediction == Y_test):.3f}")

model = LogisticRegression(random_state=42, max_iter=3000,
                           penalty='l1', solver='liblinear',
                           C=0.23357214690901212)
model.fit(X_train,Y_train)
train_predict = model.predict_proba(X_train)[:,1]
Y_train=Y_train.replace(to_replace="low_bike_demand",value=0)
Y_train=Y_train.replace(to_replace="high_bike_demand",value=1)

precision, recall, thresholds = precision_recall_curve(Y_train,
                                                       train_predict)

f1_scores = 2 * (precision * recall) / (precision + recall)
f1_scores = np.nan_to_num(f1_scores)  # Handle division by zero

best_index = np.argmax(f1_scores)
best_threshold = thresholds[best_index]

print(f"Best Threshold: {best_threshold}")
print(f"Precision: {precision[best_index]:.2f}")
print(f"Recall: {recall[best_index]:.2f}")
print(f"F1 Score: {f1_scores[best_index]:.2f}")

y_pred_thresholded = (train_predict >= best_threshold).astype(int)

from sklearn.metrics import classification_report
print(classification_report(Y_train, y_pred_thresholded))

#LDA hyperparameter grid search tuning
#errors in output caused by incompatible solvers and penalties
from sklearn.model_selection import GridSearchCV

param_grid = {
    'shrinkage': [None, 'auto'] + [x / 10 for x in range(0, 11)],
    'solver': ['svd', 'lsqr', 'eigen'],
    'store_covariance': [True, False]
}

grid_search = GridSearchCV(
    estimator=LinearDiscriminantAnalysis(),
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train, Y_train)
Y_pred = grid_search.predict(X_valid)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

#LDA
lda_model = LinearDiscriminantAnalysis(shrinkage='auto',
                                       solver='lsqr',
                                       store_covariance=True)
lda_model.fit(X_train,Y_train)
```

```
print(lda_model)
lda_train_predict = lda_model.predict_proba(X_train)
lda_prediction = np.empty(len(X_train),dtype=object)
lda_prediction = np.where(lda_train_predict[:,0]>=treshold,
                          'high_bike_demand','low_bike_demand')
print("Train Accuracy:",f"{np.mean(lda_prediction == Y_train):.3f}")

lda_test_predict = lda_model.predict_proba(X_test)
lda_prediction = np.empty(len(X_test),dtype=object)
lda_prediction = np.where(lda_test_predict[:,0]>=treshold,
                          'high_bike_demand','low_bike_demand')
print(pd.crosstab(lda_prediction,Y_test))
print("Test Accuracy:",f"{np.mean(lda_prediction == Y_test):.3f}")

Y_train=Y_train.replace(to_replace="low_bike_demand",value=0)
Y_train=Y_train.replace(to_replace="high_bike_demand",value=1)
from sklearn.metrics import classification_report
print(classification_report(Y_train, y_pred_thresholded))

#QDA hyperparameter grid search tuning
#To avoid problems due to the variables being collinear
#we use PCA to decompose data as suggested here:
#https://mevik.net/work/publications/understanding_collinearity.pdf
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_valid_pca = pca.fit_transform(X_valid)
X_test_pca = pca.fit_transform(X_test)

from sklearn.model_selection import GridSearchCV

param_grid = {
    'reg_param': [0.1, 0.2, 0.3, 0.4, 0.5,
                  0.6, 0.7, 0.8, 0.9, 0.95, 1],
    'store_covariance': [True, False]
}

grid_search = GridSearchCV(
    estimator=QuadraticDiscriminantAnalysis(),
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train_pca, Y_train)
Y_pred = grid_search.predict(X_valid_pca)
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

#QDA
qda_model = QuadraticDiscriminantAnalysis(reg_param=0.9,
                                          store_covariance=True)
qda_model.fit(X_train_pca,Y_train)
print(qda_model)
qda_train_predict = qda_model.predict_proba(X_train_pca)
qda_prediction = np.empty(len(X_train_pca),dtype=object)
qda_prediction = np.where(qda_train_predict[:,0]>=treshold,
                          'high_bike_demand','low_bike_demand')
print("Train Accuracy:",f"{np.mean(qda_prediction == Y_train):.3f}")

qda_test_predict = qda_model.predict_proba(X_test_pca)
qda_prediction = np.empty(len(X_test_pca),dtype=object)
qda_prediction = np.where(qda_test_predict[:,0]>=treshold,
                          'high_bike_demand','low_bike_demand')
print(pd.crosstab(qda_prediction,Y_test))
```

```
print("Test Accuracy:",f"{np.mean(qda_prediction == Y_test):.3f}")

Y_train=Y_train.replace(to_replace="low_bike_demand",value=0)
Y_train=Y_train.replace(to_replace="high_bike_demand",value=1)
from sklearn.metrics import classification_report
print(classification_report(Y_train, y_pred_thresholded))

# -*- coding: utf-8 -*-
"""eda_trees_and_bagging.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1-db_K7CVLyGVuzhyYD364FTVpjJdqeV8
"""

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

from google.colab import drive
drive.mount('/content/gdrive')


data = pd.read_csv('/content/gdrive/MyDrive/SML_project/training_data_fall2024.csv')
data.head()


data.groupby('increase_stock').size().plot(kind='barh',
                                           color=sns.palettes.mpl_palette('Dark2'))
plt.gca().spines[['top', 'right',]].set_visible(False)


stock_counts = data.groupby(['hour_of_day',
                             'increase_stock']).size().unstack(fill_value=0)
plt.figure(figsize=(12, 6))
stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6), colormap='viridis',
                  edgecolor='black')
plt.title('Stock Increase Categories by Hour of Day', fontsize=16)
plt.xlabel('Hour of Day', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()


stock_counts = data.groupby(['month',
                             'increase_stock']).size().unstack(fill_value=0)
plt.figure(figsize=(12, 6))
stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6), colormap='viridis',
                  edgecolor='black')
plt.title('Stock Increase Categories by Month', fontsize=16)
plt.xlabel('Month', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
stock_counts = data.groupby(['day_of_week',
                             'increase_stock']).size().unstack(fill_value=0)
plt.figure(figsize=(12, 6))
stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6), colormap='viridis',
                  edgecolor='black')
plt.title('Stock Increase Categories by Day of Week', fontsize=16)
plt.xlabel('Day of Week', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()



# Group and count data
stock_counts = data.groupby(['weekday',
                             'increase_stock']).size().unstack(fill_value=0)

# Plotting
plt.figure(figsize=(12, 6))
ax = stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6),
                       colormap='viridis', edgecolor='black')

# Adding title and labels
plt.title('Stock Increase Categories by Weekday', fontsize=16)
plt.xlabel('Weekday', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

# Adding ratio annotations
for container in ax.containers:
    for bar in container:
        # Get the height of the bar (individual count)
        height = bar.get_height()
        if height > 0:  # Only annotate non-zero bars
            # Get the x position and total for the group
            total = stock_counts.sum(axis=1)[int(bar.get_x() + 0.5)]
            # Calculate the ratio
            ratio = height / total
            # Annotate the bar with the ratio
            ax.text(
                bar.get_x() + bar.get_width() / 2,
                bar.get_height() / 2,
                f'{ratio:.2%}',  # Format as a percentage
                ha='center',
                va='center',
                fontsize=10,
                color='white'
            )

# Show the plot
plt.show()


# Group and count data
stock_counts = data.groupby(['holiday',
                             'increase_stock']).size().unstack(fill_value=0)

# Plotting
plt.figure(figsize=(12, 6))
```

```
ax = stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6),
                       colormap='viridis', edgecolor='black')

# Adding title and labels
plt.title('Stock Increase Categories by Holiday', fontsize=16)
plt.xlabel('Holiday', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

# Adding ratio annotations
for container in ax.containers:
    for bar in container:
        # Get the height of the bar (individual count)
        height = bar.get_height()
        if height > 0:  # Only annotate non-zero bars
            # Get the x position and total for the group
            total = stock_counts.sum(axis=1)[int(bar.get_x() + 0.5)]
            # Calculate the ratio
            ratio = height / total
            # Annotate the bar with the ratio
            ax.text(
                bar.get_x() + bar.get_width() / 2,
                bar.get_height() / 2,
                f'{ratio:.2%}',  # Format as a percentage
                ha='center',
                va='center',
                fontsize=10,
                color='white'
            )

# Show the plot
plt.show()


ddata = data.drop(['snow', 'summertime'], axis = 1)
ddata.describe()


grouped_data = data.groupby(['month'])[['temp', 'precip', 'snowdepth',
                                        'humidity', 'dew' ,'windspeed',
                                        'cloudcover', 'visibility']].mean()
print(grouped_data)


data.dtypes


for col in data.columns:
  print(col, len(data[col].unique()))


#data_with_dummies = pd.get_dummies(ddata, columns=['increase_stock'], prefix='stock', drop_f
#data_with_dummies = data_with_dummies.drop(['increase_stock'], axis = 1)
ddata['increase_stock'] = ddata['increase_stock'].replace({
    'low_bike_demand': 0,
    'high_bike_demand': 1
})
ddata.head()


#Correlation matrix
plt.figure(figsize=(12, 8))
```

```
sns.heatmap(ddata[['holiday', 'weekday', 'temp', 'dew', 'humidity', 'precip',
                   'snowdepth', 'windspeed', 'cloudcover', 'visibility',
                   'increase_stock']].corr(), annot=True, cmap='coolwarm',
                    fmt=".2f")

plt.title('Correlation Matrix', fontsize=16)
plt.show()


holiday_data = ddata[(ddata['holiday'] == 1)]
non_holiday_data = ddata[(ddata['holiday'] == 0)]
ddata.head()
print(len(holiday_data[(holiday_data['increase_stock'] == 0)])/len(
    holiday_data[(holiday_data['increase_stock'] == 1)]))
print(len(non_holiday_data[(non_holiday_data['increase_stock'] == 0)])/len(
    non_holiday_data[(non_holiday_data['increase_stock'] == 1)]))


(ddata['weekday'] - ddata['holiday']).sum()


ddata.describe()


(ddata['precip'] > 0)


data['rainy_day'] = (data['precip'] > 0).astype(int)
data['snowy_day'] = (data['snowdepth'] > 0).astype(int)



# Group and count data
stock_counts = data.groupby(['snowy_day',
                             'increase_stock']).size().unstack(fill_value=0)

# Plotting
plt.figure(figsize=(12, 6))
ax = stock_counts.plot(kind='bar', stacked=True, figsize=(12, 6),
                       colormap='viridis', edgecolor='black')

# Adding title and labels
plt.title('Stock Increase Categories by Snowy Day', fontsize=16)
plt.xlabel('Snowy Day', fontsize=14)
plt.ylabel('Count of Increase Stock', fontsize=14)
plt.xticks(rotation=0)
plt.legend(title="Increase Stock Category", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

for container in ax.containers:
    for bar in container:
        height = bar.get_height()
        if height > 0:  # Only annotate non-zero bars
            # Get the x position and total for the group
            total = stock_counts.sum(axis=1)[int(bar.get_x() + 0.5)]
            ratio = height / total
            print (ratio)
            # Annotate the bar with the ratio
            ax.text(
                bar.get_x() + bar.get_width() / 2,
                bar.get_height() / 2,
                f'{ratio:.2%}',  # Format as a percentage
                ha='center',
                va='center',
```

```
                fontsize=10,
                color='white'
            )

# Show the plot
plt.show()


data=data.replace(to_replace="low_bike_demand",value=0)
data=data.replace(to_replace="high_bike_demand",value=1)
# data.head()
precip_data = data[(data['snowdepth'] > 0)]
non_precip_data = data[(data['snowdepth'] == 0)]
data.head()
print(len(precip_data[(precip_data['increase_stock'] == 0)])/len(
    precip_data))
print(len(non_precip_data[(non_precip_data['increase_stock'] == 0)])/len(
    non_precip_data))


from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.metrics import classification_report, accuracy_score, f1_score


data['increase_stock'] = data['increase_stock'].replace({
   'low_bike_demand': 0,
   'high_bike_demand': 1
   })


X = data.drop(columns=['increase_stock', 'summertime', 'snow'])
y = data['increase_stock']

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1,
                                                            random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.111, random_state=42)


tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train, y_train)

y_pred = tree_model.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


best_model = None
best_f1_score = 0

for depth in [3, 5, 10, None]:
    for min_samples_split in [2, 5, 10]:
        for min_samples_leaf in [1, 2, 5]:
            tree_model = DecisionTreeClassifier(
                max_depth=depth,
                min_samples_split=min_samples_split,
                min_samples_leaf=min_samples_leaf,
                random_state=42
            )
            tree_model.fit(X_train, y_train)
            y_pred = tree_model.predict(X_val)
            # F1_score for 'high_bike_demand'
```

```
            f1_class_1 = f1_score(y_val, y_pred, average='weighted')
            score = accuracy_score(y_val, y_pred)

            if f1_class_1 > best_f1_score:
                best_f1_score = f1_class_1
                best_score = score
                best_model = tree_model
                best_params = {
                    'max_depth': depth,
                    'min_samples_split': min_samples_split,
                    'min_samples_leaf': min_samples_leaf
                }

print("Best score:", best_score)
print("Best F1 score:", best_f1_score)
print("Best Parameters:", best_params)


from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5],
    'criterion': ['gini', 'entropy']
}

grid_search = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=42),
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train, y_train)
y_pred = grid_search.predict(X_val)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = grid_search.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


data.head()


# Sunrise and Sunset times (rounded to nearest hour)
sunrise_sunset = {
    1: {'sunrise': 7, 'sunset': 17},
    2: {'sunrise': 7, 'sunset': 18},
    3: {'sunrise': 7, 'sunset': 19},
    4: {'sunrise': 6, 'sunset': 20},
    5: {'sunrise': 5, 'sunset': 20},
    6: {'sunrise': 5, 'sunset': 21},
    7: {'sunrise': 6, 'sunset': 21},
    8: {'sunrise': 6, 'sunset': 20},
```

```
    9: {'sunrise': 6, 'sunset': 19},
    10: {'sunrise': 7, 'sunset': 18},
    11: {'sunrise': 7, 'sunset': 17},
    12: {'sunrise': 7, 'sunset': 17}
}


# Function to determine if it's day based on hour_of_day
def determine_day(row):
    sunrise = sunrise_sunset[row['month']]['sunrise']
    sunset = sunrise_sunset[row['month']]['sunset']
    if sunrise <= row['hour_of_day'] < sunset:
        return 1
    else:
        return 0


data['day'] = data.apply(determine_day, axis=1)


mean_humidity = data['humidity'].mean()
mean_temp = data['temp'].mean()


data['good_weather_humid_temp'] = 0  # Initialize the column with 0s
data.loc[(data['humidity'] < mean_humidity) & (data['temp'] < mean_temp),
         'good_weather_humid_temp'] = 1


data['rainy_day'] = (data['precip'] > 0).astype(int)
data['snowy_day'] = (data['snowdepth'] > 0).astype(int)
data['good_weather_rain_snow'] = 0
data.loc[(data['rainy_day'] == 0) & (data['snowy_day'] == 0),
         'good_weather_rain_snow'] = 1


X = data.drop(columns=['increase_stock', 'summertime', 'snow'])
y = data['increase_stock']

# Stratify maintains the same class distribution in both train and test
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, st
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1,
                                                            random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.111, random_state=42)


tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train, y_train)

y_pred = tree_model.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5],
    'criterion': ['gini', 'entropy']
}
```

```
grid_search = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=42),
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train, y_train)
y_pred = grid_search.predict(X_val)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = grid_search.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


importances = tree_model.feature_importances_
feature_names = X_train.columns
feature_importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
})

feature_importance_df = feature_importance_df.sort_values(by='Importance',
                                                          ascending=False)

plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'],
         color='skyblue')
plt.xlabel('Importance')
plt.title('Feature Importance - Decision Tree')
plt.tight_layout()
plt.show()



prime_features = ['hour_of_day', 'temp', 'humidity', 'windspeed', 'dew',
                  'cloudcover', 'month', 'day_of_week', 'precip', 'visibility',
                  'holiday']

X = data[prime_features]
y = data['increase_stock']

# Stratify maintains the same class distribution in both train and test

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1,
                                                            random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.111, random_state=42)

tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train, y_train)

y_pred = tree_model.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
```

```
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = tree_model.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


# prime_features = ['hour_of_day', 'temp', 'humidity', 'windspeed', 'dew',
# 'cloudcover', 'month', 'day_of_week', 'precip', 'visibility', 'holiday']

X = data.drop(columns=['increase_stock', 'summertime', 'snow'])
y = data['increase_stock']

# Stratify maintains the same class distribution in both train and test

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1,
                                                            random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.111, random_state=42)

rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)

y_pred = rf_model.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = rf_model.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


param_grid = {
    'n_estimators': [10, 50, 100, 200],  # Number of trees in the forest
    'max_depth': [None, 10, 20, 30],  # Maximum depth of each tree
    'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split a node
    'min_samples_leaf': [1, 2, 4],    # Minimum number of samples required to be at a leaf noa
    'bootstrap': [True, False]        # Whether to use bootstrap samples when building trees
}

grid_search = GridSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)

grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

y_pred = grid_search.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))
```

```
y_pred = grid_search.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


importances = rf_model.feature_importances_
feature_names = X_train.columns
feature_importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
})

feature_importance_df = feature_importance_df.sort_values(by='Importance',
                                                          ascending=False)

plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'],
         color='skyblue')
plt.xlabel('Importance')
plt.title('Feature Importance - Decision Tree')
plt.tight_layout()
plt.show()


X = data.drop(columns=['increase_stock', 'summertime', 'snow'])
y = data['increase_stock']

# Stratify maintains the same class distribution in both train and test
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1,
                                                            random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.111, random_state=42)

bag_model = BaggingClassifier(random_state=42)
bag_model.fit(X_train, y_train)

y_pred = bag_model.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = bag_model.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))


param_grid = {
    'n_estimators': [10, 50, 100, 200],  # Number of base estimators (trees)
    'max_samples': [0.5, 0.8, 1.0],    # Proportion of samples to train each base estimator on
    'max_features': [0.5, 0.8, 1.0],  # Proportion of features to train each base estimator on
    'bootstrap': [True, False],        # Whether to use bootstrap samples
    'n_jobs': [-1]                     # Use all available processors
}

grid_search = GridSearchCV(
    estimator=BaggingClassifier(random_state=42),
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,  # 5-fold cross-validation
    n_jobs=-1,  # Use all available processors
    verbose=1
)
```

```
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

y_pred = grid_search.predict(X_val)

print("Classification Report:\n", classification_report(y_val, y_pred))
print("Accuracy Score:", accuracy_score(y_val, y_pred))


y_pred = grid_search.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Test Accuracy:", accuracy_score(y_test, y_pred))

# -*- coding: utf-8 -*-
"""knn_project-2.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/16xsEEV257Hl2-nQGAcXAhiQF-p-m5jgr
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.decomposition import PCA
from sklearn.inspection import DecisionBoundaryDisplay

# Load the dataset
data = pd.read_csv("training_data_fall2024.csv")
X = data.drop(columns=['increase_stock', 'summertime', 'snow'])
y = data['increase_stock']


label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)



X_train_val, X_test, y_train_val, y_test = train_test_split(X, y_encoded, test_size=0.1, randc
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,test_size=0.111, ra



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

# Initialize and train the KNN classifier
knn = KNeighborsClassifier(n_neighbors=11, weights='distance')
knn.fit(X_train_scaled, y_train)


y_pred_train = knn.predict(X_train_scaled)
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")
print("\nClassification Report on training data:\n", classification_report(y_train, y_pred_tra
```

```
print("\nConfusion Matrix on training data:\n", confusion_matrix(y_train, y_pred_train))

# Initialize and train the KNN classifier
knn = KNeighborsClassifier(n_neighbors=11, weights='uniform')
knn.fit(X_train_scaled, y_train)


y_pred = knn.predict(X_test_scaled)
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))

from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import numpy as np
import pandas as pd

# Initialize an empty list to store results
results = []

# Loop through different values of k and weight configurations
for k in [3, 5, 8, 11]:
    for weight in ['uniform', 'distance']:
        # Initialize KNeighborsClassifier
        knn = KNeighborsClassifier(n_neighbors=k, weights=weight)


        cv_scores = cross_val_score(knn, X_train_scaled, y_train, cv=5, scoring='accuracy')


        knn.fit(X_train_scaled, y_train)

        # Predict on training and testing sets
        y_pred_train = knn.predict(X_train_scaled)
        y_pred_test = knn.predict(X_test_scaled)


        train_accuracy = accuracy_score(y_train, y_pred_train)


        test_accuracy = accuracy_score(y_test, y_pred_test)

        # Calculate cross-validation accuracy (mean of cross-validation scores)
        cv_accuracy = np.mean(cv_scores)

        # Store the results
        results.append({
            'k': k,
            'weights': weight,
            'Training Accuracy': train_accuracy * 100,
            'Testing Accuracy': test_accuracy * 100,
            'Cross-Validation Accuracy': cv_accuracy * 100
        })


results_df = pd.DataFrame(results)


print(results_df)

# -*- coding: utf-8 -*-
"""boost.ipynb

Automatically generated by Colab.
```

```
Original file is located at
    https://colab.research.google.com/drive/1RH4t9QVhCcfBQuebqopBKH5W-afopxnL

<h1>BOOSTING BASED MODELS</h1>
"""

from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.ensemble import GradientBoostingClassifier, AdaBoostClassifier #Using due to less
from sklearn.preprocessing import StandardScaler
import xgboost as xgb
from sklearn.model_selection import GridSearchCV, StratifiedKFold, cross_val_score

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.1, random_state
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.111, r

"""<h2> ADABOOST </h2>

<b>Case 1: Default Base Estimator</b>
Basic decision tree is used. 20 different quantaties of base estimators were tried. Additional
"""

param_grid = {
    'n_estimators': [5, 10, 20, 30, 50],
    'learning_rate': [0.5, 0.6, 0.7, 0.8, 0.9, 1]
}
adaboost = AdaBoostClassifier()
grid_search = GridSearchCV(estimator=adaboost, param_grid=param_grid, cv=10, scoring='accuracy
grid_search.fit(X_train_val, y_train_val)
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)

#BEST MODEL
#Creating booster
ada = AdaBoostClassifier(n_estimators=10, learning_rate = 0.6)
#Training on data
best_ada = ada.fit(X_train, y_train)
#RESULTS
train_pred_ada = best_ada.predict(X_train)
print(metrics.confusion_matrix(y_train, train_pred_ada))
print(metrics.accuracy_score(y_train, train_pred_ada))
val_pred_ada = best_ada.predict(X_val)
print(metrics.confusion_matrix(y_val, val_pred_ada))
print(metrics.accuracy_score(y_val, val_pred_ada))
#TEST RESULT
test_pred_ada = best_ada.predict(X_test)
print(metrics.confusion_matrix(y_test, test_pred_ada))
print(metrics.accuracy_score(y_test, test_pred_ada))

"""<h2>XGBOOST</h2>"""

#Fitting the model
model = xgb.XGBClassifier()
model.fit(X_train, y_train)
# Calculate training accuracy
y_train_pred = model.predict(X_train)
train_accuracy = metrics.accuracy_score(y_train, y_train_pred)
print(f"Training Accuracy: {train_accuracy:.4f}")
#Perform 10-fold cross-validation on validation data
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
cross_val_accuracies = cross_val_score(model, X_val, y_val, cv=kf, scoring="accuracy")
mean_cross_val_accuracy = cross_val_accuracies.mean()
print(f"10-Fold Cross-Validation Accuracy: {mean_cross_val_accuracy:.4f}")

"""The initial XGBoost model was found to overfit the data. Therefore it was considered to use
```

```
#Performing grid search for
param_grid = {
    'reg_lambda': [1, 5, 10],     # L2 regularization
    'gamma': [1, 5, 10],      # Minimum loss reduction
    'n_estimators' : [10, 50, 100],
    'learning_rate' : [0.1, 0.5, 1.0]
    }
#Initialising model
xgb_model = xgb.XGBClassifier(objective='binary:logistic', random_state=42)
#Performing grid search
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=10, scoring='accurac
grid_search.fit(X_train_val, y_train_val)

print("Best Parameters:", grid_search.best_params_)
print("Best CV Accuracy:", grid_search.best_score_)
best_model = grid_search.best_estimator_

#BEST MODEL
#Creating booster
xgb_model = xgb.XGBClassifier(objective = 'binary:logistic', random_state = 42, n_estimators=7
#Training on data
best_xg = xgb_model.fit(X_train, y_train)
#RESULTS
train_pred_xg = best_xg.predict(X_train)
print(metrics.confusion_matrix(y_train, train_pred_xg))
print(metrics.accuracy_score(y_train, train_pred_xg))
val_pred_xg = best_ada.predict(X_val)
print(metrics.confusion_matrix(y_val, val_pred_xg))
print(metrics.accuracy_score(y_val, val_pred_xg))
#TEST RESULT
test_pred_xg = best_ada.predict(X_test)
print(metrics.confusion_matrix(y_test, test_pred_xg))
print(metrics.accuracy_score(y_test, test_pred_xg))
```