Statistical Machine Learning Project 2023

Niclas Svensson
niclasdan@gmail.com

Abstract

In this project several machine learning algorithms were trained and tested based on data collected over three years from Washington D.C. The data included parameters such as the snow depth, days of the week, amount of rain etc. Through feature engineering and data analysis the data was cleaned up and deployed to train the algorithms. After tuning and evaluation, the random forest algorithm scored the highest on the chosen metrics, accuracy and f1 score. Based on the findings made in this report, further testing will be done using the random forest algorithm.

1 Introduction to the problem

This project focuses on the capital bikeshare system in Washington D.C, where the availability of bikes often does not meet the demand. This problem leads to potential increase in CO2 emissions as people may chose public transport or cars over bicycles.

The group is faced with the challenge to predict whether an increase in available bikes at an certain hour is necessary using different types of data. Ultimately, the goal is to select the best performing machine learning model to put against the test data.

2 Data analysis

Prior to commencing model development, we conducted a comprehensive examination of the data utilizing a tool known as ydata_profiling. This analysis was instrumental in gaining insights into the distribution of each data point. Notably, it highlighted anomalies, such as the 'snow' feature containing solely 0 values, rendering it non-contributory to our model's learning process. Further elaboration on this will be provided in the Feature Engineering section.

Moreover, this tool facilitated the identification of crucial data segments and insignificantly impactful ones. Additionally, it offered a visualization of data interconnections, aiding in strategic decisions regarding feature enhancement in subsequent stages.

The Categorical inputs were: hour_of_day, day_of_week, month, holiday, weekday, summertime, snow and increase_stock.
The numerical inputs were: temp, dew, humidity, precip, snowdepth, windspeed, cloudcover and visibility.

2.1 Feature engineering

We replaced "month" with *is_winter*, *is_spring*, *is_summer* and *is_fall* to capture a broader weather related impact on bike usage. This was done by mapping each month to it's respective season effectively making it a binary output instead of a numerical.

Introducing *rush_hour* and *night_time* which targets the important times of day with varying bike usage. *Rush_hour* being morning and afternoon and *night_time* being late night and early morning. This binary categorization helps distinguish between peak demand times of the day.

Converting *snowdepth* to a binary *is_there_snow* feature simplifies the model by focusing on the presence of snow rather than the amount. By transforming it into a binary feature we could simply ask: does snow affect bike demand?

Changing the target variable to a binary *increase_stock* will label it under one category making it easier to process the data with our machine learning algorithms because it's a binary output under one category.

Continuous variables such as *temp*, *humidity*, *windspeed*, *cloudcover* and *visibility* were normalized using StandardScaler. This is done to ensure that all these features contribute equally to the final result without any single feature dominating the rest due to it's scale.

In summary we dropped the month column for *is_winter*, *is_spring*, *is_summer* and *is_fall* to define seasons instead. Then we add *rush_hour* and *night_time* from *hour_of_day* to get categorical features for important times during the day. The numerical feature *snowdepth* was also dropped for *is_there_snow* to get a categorical feature instead. Lastly, we also dropped *summertime*, *day_of_week*, *dew*, *precip* and *snow*.

After we managed our features in an appropriate way we started inspecting our data.

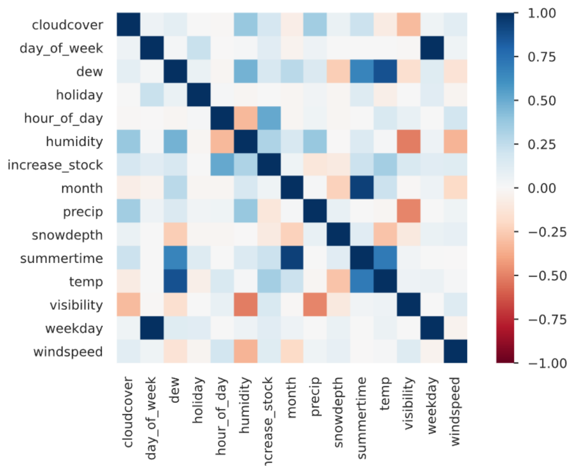

Figure 1: Correlations between different input features

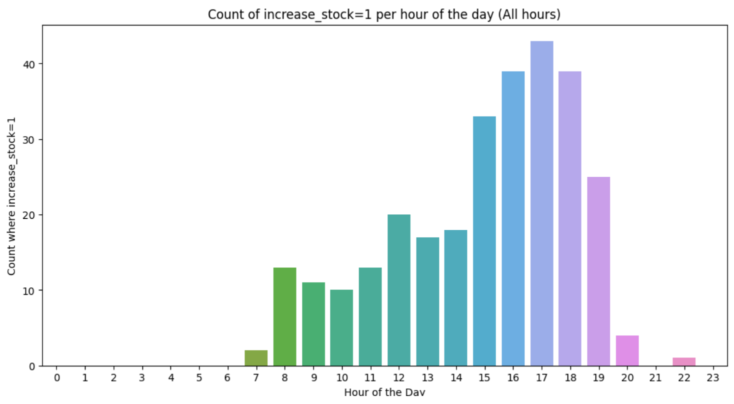

Figure 2: Bike demand over different hours of the day

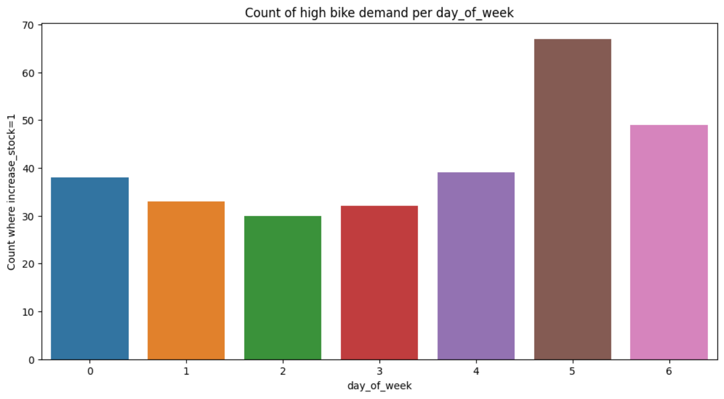

Figure 3: Bike demand over different days of the week

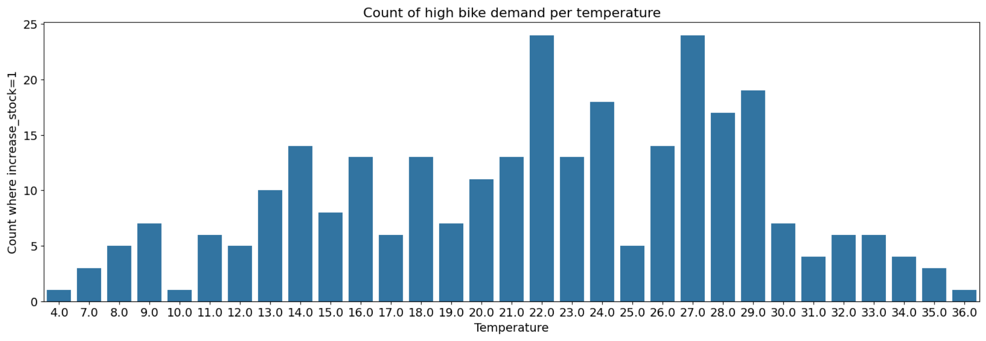

Figure 4: Bike demand over different temperatures

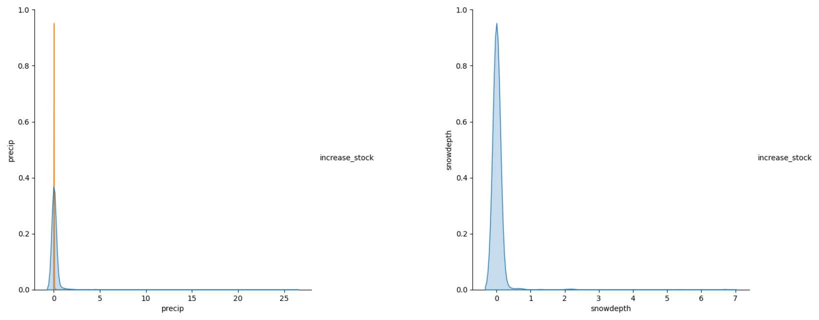

Figure 5: Bike Demand depending on Precipitation(left)/Snowdepth(right)

Figure 3 depict the bike demand over a day. Here we can see that independent of the day, around 15 to 19 seem to have the highest bike demand. By inspecting figure 4 we see that weekends have higher demand than weekdays. Based on Figure 5 and 6 we can see weather has an effect on the bike demand. Figure 6 clearly shows that if there is precipitation or if there "is" a snowdepth the bike demand is almost always zero, regardless of the amount of precipitation or magnitude of the snowdepth. Figure 5 displays that the bike demand increases as the weather is above 12 degrees, but decreases rapidly if we surpass 29 degrees.

3 Model development

3.1 K-nn classifier

The data set is composed of a feature matrix X and a target variable y. The feature matrix X includes all the independent variables while the target variable y represents the binary outcome of high or low bike demand. Each feature in X is then normalized to ensure equal contribution to the result. Without normalization different features might scale differently and influence the model. For the k-nn algorithm we classify data points based on its neighbors. It operates by finding the k nearest neighbors and assigns the most frequent label among these neighbors to the test point. The k value of the k-nn algorithm decides whether it functions more towards bias or variance. A low k might lead to the model reacting to strongly to small fluctuations while a high k might cause the model to miss out on small but important pieces of information. Therefore, identifying the optimal k value involves extensive amount of testing.

$$f(x; k, T) = \arg\max_{y} \sum_{i=1}^{n} w(x, X_i; k)\mathbf{1}\{y_i = y\} \tag{1}$$

Eq (1) is the mathematical concept behind the K-nn method. The classification function for a new point x given the number of neighbors k and the training data T. The weight function depends on x, the training data point Xi and the numbers of neighbors k considered. Lastly we have the indicator function which outputs 1 if yi is equal to y and 0 otherwise.

$$d(P, Q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

To calculate the distance between neighbors and input points we use Euclidean distance. In 2 space we use x and y coordinates to define the distance between neighbors and add more parameters if needed. All the parameters are then squared to find the straight line distance between two neighbors.

3.2 Discriminant Analysis

Discriminant analysis can be used to predict the labels of the output for new input data. Thus, discriminant analysis aims to estimate $p(y|x)$. The classifier which is obtained after applying the joint probability distribution function, as well as some expressions taken from the Gaussian mixture model (GMM) is:

$$p(y = m|\mathbf{x}_*) = \frac{\hat{\pi}_m N(\mathbf{x}_*|\hat{\mu}_m, \hat{\Sigma}_m s)}{\Sigma_{j=1}^{M} \hat{\pi}_j N(x_*|\hat{\mu}_j, \hat{\Sigma}_j)} \tag{3}$$

The parameters to be trained are $\theta = \{\mu_m, \Sigma_m, \pi_m\}_{m=1}^{M}$. where μ_m, Σ_m and π_m is the mean vector, covariance matrix and the marginal class probabilities for each class. The mean vector of each class is given by:

$$\hat{\mu}_m = \frac{1}{n_m} \sum_{i:y_i=m} x_i \tag{4}$$

And the covariance matrix for the different classes are estimated as:

$$\hat{\sum}_m = \frac{1}{n_m}(x_i - \hat{\mu}_m)(x_i - \hat{\mu}_m)^T \tag{5}$$

The marginal class probabilities for each class is $\hat{\pi_m} = \frac{n_m}{n}$. The 'hard' predictions ($\hat{y}_*$) can be obtained, by choosing the class which has the most probable prediction with:

$$\hat{y_*} = \arg\max_{m} p(y = m|\mathbf{x}_*) \tag{6}$$

As the decision boundaries of this classifier takes on a quadratic form, this classifier approach is denoted as Quadratic Discriminant Analysis (QDA). However, if the assumption is made that the covariance matrix is the same for all of the classes, the Linear Discriminant Analysis (LDA) is obtained instead. (1) LDA and QDA both assumes that the dataset is drawn from a Gaussian distribution. The assumption of all the covariance matrices being the same results in a dimensionality reduction of the dataset, where LDA projects the input data onto a linear subspace which is used to classify the output labels. As QDA does no assumption about the covariance between classes in the dataset, the parameter it handles is increased. Because of this QDA has an amount of parameters which is quadratic in regards of the amount of input. This results in QDA having its quadratic decision boundary.(2)

3.3 Tree Based Method

Tree based methods are supervised learning techniques, that are adaptable for both classification and regression problems. They can be used for both linear and non-linear problems.

3.3.1 Decision Trees

Often referred to as rule-based models, they divide the input space into multiple region, based on the "rules" imposed on certain features. These ordered rules can be represented as a tree structured graph, hence the name. With this model, we are looking for the prediction $\hat{y}(x_*)$, such as:

$$\hat{y}(\mathbf{x}_*) = \sum_{l=1}^{L} \hat{y}_l I\{\mathbf{x}_* \in R_l\} \tag{7}$$

where L denotes the number of regions, R_l is the l-th region and $\hat{y}_l$ is the prediction for the region by the model. $I\{x_* \in R_l\}$ is 1 if the point is in the given region, 0 otherwise. By finding suitable $\hat{y}$ values and R regions, we can teach our model to classify data. We use recursive binary splitting to do this, a greedy algorithm that tries to minimize the cost function after each split. Two commonly used cost functions are the squared error or an optimization problem defined as:

$$argmin_{nextsplit}(n_1 Q_1 + n_2 Q_2) \tag{8}$$

Here n_i is the number of samples in the i-th region, while Q is a function that measures the quality of the split, often called the criterion. In our solution we used the *Gini-index*, which is defined as:

$$Q_l = \sum_{m=1}^{M} \hat{\pi}_{lm}(1 - \hat{\pi}_{lm}) \tag{9}$$

where $\hat{\pi}_{lm}$ is the ratio of training observations within the l-th region that are part of the m-th class.

By defining this metrics, we can start building the classification tree. To do this, we divide the input space into two regions, based on one of the selected features from the input and the loss measuring function. Once we have the two regions, defined by the cut point s:

$$R_1(i,s) = \{\mathbf{x}|x_i < s\} \quad and \quad R_2(i,s) = \{\mathbf{x}|x_i > s\} \tag{10}$$

We repeat this step on the regions, until a certain depth, or some kind of other criterion is reached.

Once we are done building the tree, we have the prediction for this regions defined, as the majority class of the points in that region:

$$\hat{y}_1 = MajorityVote\{y_i : \mathbf{x_i} \in R_1\} \quad and \quad \hat{y}_2 = MajorityVote\{y_i : \mathbf{x_i} \in R_2\} \tag{11}$$

3.4 Random Forest

Ensemble techniques create a single, more accurate model using several base ones. Random forests have decision trees, as their base models. We bootstrap the data to create multiple data sets and different decision trees are then trained on these. Each tree is utilizing only a subset of inputs. This further injections randomness into the structure helps prevent any dominant set of inputs from consistently winning in every tree, making this more effective than bagging. Through this technique, we successfully reduce the correlation between different trees, resulting in a significant reduction in variance when averaging out their votes. During training we can grow each tree in parallel and control their attributes such as height, splits, number of trees, with hyperparameters. Each tree is trained on a bootstrapped dataset using selected subsets of features, so it is a computationally heavy model, however it produces better results than single decision trees.

3.5 Logistic Regression

Logistic regression offers a method to model conditional class probabilities. It's essentially an adaptation of the linear regression model tailored to address classification problems rather than regression. It serves as a statistical approach for binary classification tasks, estimating the probability of a binary outcome based on input features.

3.5.1 Mathematical model

The aim is to find a function $g(x)$ that estimates the likelihood of the positive class. Initially, we employ the linear regression model, expressed as

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_p x_p = \theta^T x \tag{12}$$

which maps x to z. Here, z spans the entire real number space, but we require a function that outputs within the range $[0, 1]$. Logistic regression addresses this by compressing z towards the interval $[0, 1]$ using the logistic function $h(z) = \frac{e^z}{1+e^z}$. This transformation yields:

$$g(x) = \frac{e^{\theta^T x}}{1 + e^{\theta^T x}} \tag{13}$$

This revised function $g(x)$ ensures values between 0 and 1, allowing for interpretation as probabilities.

For logistic regression, training the model involves finding the optimal parameters that minimize the logistic loss.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{14}$$

For solving the problem `sklearn.linear_model.LogisticRegression`, the `liblinear` solver has been used. The `liblinear` solver is a numerical optimization algorithm designed for logistic regression and linear support vector machines (SVMs). It employs an iterative numerical optimization technique, typically a coordinate descent algorithm.

3.6 Hyperparameters

The tuning of the various models was conducted with GridSearchCV. For logistic regression model we covered parameters such as the regularization strength, penalty type, solver algorithms, maximum iterations etc. The hyperparameter tuned for LDA was the solver, resulting in the best solver being svd. The tuned hyperparameter for QDA was the regularization parameter, which regularizes the estimates per-class covariance by transforming S2, which corresponds to the scaling attributes of the different classes. Random forest was more difficult to tune as it tended to overfit the data when using few trees. The alteration of the number of trees, their depth and number of samples per tree showed to have the most impact. For k-NN the tuned hyperparameter was the amount of neighbors (k) to include for the test value. The optimal k was found using for-loops.

3.7 Evaluation and performance

For the naive model we used (`sklearn.dummy.DummyClassifier`) that serves as a simple baseline to compare against our other more complex models. It generates predictions uniformly at random from the list of unique classes, so each class has equal probability.

The data set was divided into training and validation sets employing a 20-fold cross-validation strategy (`sklearn.model_selection.KFold`). This method splits the data set into subsets, trains the model on various combinations, and computes the metrics. The mean value of each metric, across validation sets provided an estimate of the model's performance. We chose to evaluate our models based on accuracy and F1 score. The accuracy would give us a peek into the general performance of the model. Precision and recall have their own reasons behind them, higher precision would mean higher customer satisfaction and less emission, while higher recall yields higher profits for the operating company. We decided on evaluating the model's F1 score, that takes both of them into account.

After training the models on the training set using k-fold cross validation, we evaluated them on the test set as well. We achieved the following results

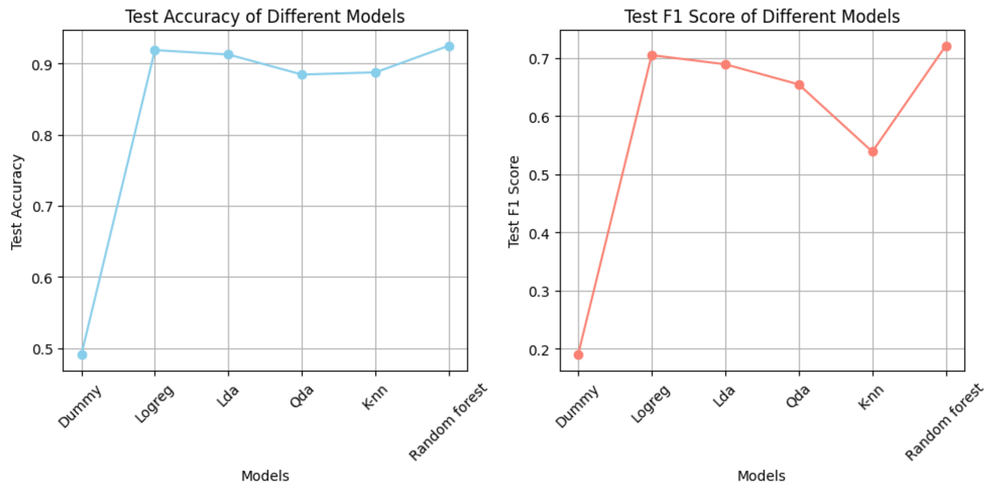

Figure 6: Accuracy and F1 scores of the models

It is easy to see that why we ended up choosing the random forest model, it performed the best including accuracy and F1 score as well before the logistic regression one.

4 Conclusions

The final model was evaluated as the best based on accuracy and F1 score, using a 20-fold cross validation strategy and further tested on a separate test set. This evaluation of each model provided insight into it's strengths and limitations for the bike-sharing system.

Furthermore, the project takes a methodical approach to machine learning in a real-world scenario. The project delved into feature engineering, implementation of different models and optimizing the hyper parameters for the best performance. This methodical and comprehensive strategy not only addresses the immediate need of the bike-sharing system but also lays a foundation for future enhancements that machine learning algorithms can perform.

References

[1] Lindholm, Andreas, Niklas Wahlström, Fredrik Lindsten, & Thomas B. Schön. (2022). *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press.

[2] Scikit-learn, *Linear and Quadratic Discriminant Analysis (2023)*, last accessed 20 december 2023, `https://scikit-learn.org/stable/modules/lda_qda.html`

A Appendix

A.1 Main code

```python
#importing all neccesary libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import sklearn.discriminant_analysis as skl_da
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 sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import f1_score
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.model_selection import GridSearchCV

np.random.seed(1)

# loading the data
data = pd.read_csv('/content/training_data.csv', dtype={'ID':
↪   str}).dropna().reset_index(drop=True)

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

# Plotting
data_snow = data[['snowdepth', 'increase_stock']]
plot_snow = sns.pairplot(data_snow, hue = 'increase_stock', height = 6);

data_precip = data[['precip', 'increase_stock']]
plot_precip = sns.pairplot(data_precip, hue = 'increase_stock', height = 6);

#do not modify this seed value, shuffle the train data in an another part of
↪   the code, so we all have the same test set

# Categorizing different variables
data['is_winter'] = data['month'].apply(lambda x: 1 if x in [12, 1, 2] else 0)
data['is_spring'] = data['month'].apply(lambda x: 1 if x in [3, 4, 5] else 0)
data['is_summer'] = data['month'].apply(lambda x: 1 if x in [6, 7, 8] else 0)
data['is_fall'] = data['month'].apply(lambda x: 1 if x in [9, 10, 11] else 0)
data['rush_hour'] = data['hour_of_day'].apply(lambda x: 1 if (7 <= x <= 9) or
↪   (17 <= x <= 19) else 0)
data['night_time'] = data['hour_of_day'].apply(lambda x: 1 if (20 <= x) or (7
↪   >= x) else 0)
data['is_there_snow'] = data['snowdepth'].apply(lambda x: 1 if (0 < x) else 0)
data['increase_stock'] = data['increase_stock'].map({'low_bike_demand': 0,
↪   'high_bike_demand': 1})

# dropping variables from the dataframe
```

```python
columns_to_drop = ['snowdepth','summertime','day_of_week','month','dew',
 'precip', 'snow']
data.drop(columns=columns_to_drop, inplace=True)

#normalizing numerical coloumns
normalized_cols = ['temp', 'humidity', 'windspeed', 'cloudcover',
 'visibility']
dummy_cols = ['hour_of_day', 'holiday', 'weekday']

scaler = StandardScaler()
data[normalized_cols] = scaler.fit_transform(data[normalized_cols])

# creating data set with all the data to train the final chosen method
x_final = data.drop(columns = ['increase_stock'])
y_final = data['increase_stock']

seed_value = 42
#Splitting the data into training set and a testing set
data, test_data = skl_ms.train_test_split(data, test_size=0.2,
 random_state=seed_value, shuffle = False)

# seperating labels and input data
x_training = data.drop(columns = ['increase_stock'])
y_training = data['increase_stock']
x_testing = test_data.drop(columns = ['increase_stock'])
y_testing = test_data['increase_stock']

# assigning models
LDA = skl_da.LinearDiscriminantAnalysis()
QDA = skl_da.QuadraticDiscriminantAnalysis()

# splitting data into training set and validation set
x_train, x_val, y_train, y_val = skl_ms.train_test_split(x_training,
 y_training, train_size = 0.80, random_state = 1 )

# fit the basic models
LDA.fit(x_train, y_train)
QDA.fit(x_train, y_train)

# accuracy on training data
LDA_score_train = LDA.score(x_train, y_train) # 0.869
QDA_score_train = QDA.score(x_train, y_train) # 0.196

# accuracy on validation data
LDA_score_val = LDA.score(x_val, y_val) # 0.859
QDA_score_val = QDA.score(x_val, y_val) # 0.164

# Tuning LDA, grid search for best solver
# tuning done on the left out validation set
# increasing accuracy to 0.898
param_grid = {'solver': ['svd', 'lsqr', 'eigen']}
grid_search = skl_ms.GridSearchCV(LDA, param_grid, cv=5)
grid_search.fit(x_val, y_val)

best_paramss = grid_search.best_params_

best_LDA = grid_search.best_estimator_

y_pred = best_LDA.predict(x_val)
accuracy = accuracy_score(y_val, y_pred) # LDA tuned result = 0.848

# Tuning QDA, grid search to optimize the regularization parameter
# tuning done on the left out validation set
# increasing accuracy to 0.887
```

```python
params = [{'reg_param': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]}]
cv = skl_ms.RepeatedStratifiedKFold(n_splits = 10, n_repeats = 3, random_state
↪  = 1)
search = skl_ms.GridSearchCV(QDA, params, cv = cv)
search.fit(x_val, y_val)


best_params = search.best_params_
best_QDA = search.best_estimator_

y_pred = best_QDA.predict(x_val)
acc = accuracy_score(y_val, y_pred)

print('Accuracy on Train data')
print(f'LDA: {LDA_score_train:.3f}')
print(f'QDA: {QDA_score_train:.3f}')
print('Val data')
print(f'LDA: {LDA_score_val:.3f}')
print(f'QDA: {QDA_score_val:.3f}')
print('Grid Search alteration')
print(f'Grid Search result on LDA {accuracy:.3f}')
print(f'Grid search result on QDA {acc:.3f}');

#train the tuned models
best_LDA.fit(x_training, y_training)
best_QDA.fit(x_training, y_training)

accuraccy_rates=[]
# Specified parameters
params = {
    'C': 1.0,
    'class_weight': None,
    'fit_intercept': True,
    'intercept_scaling': 1,
    'max_iter': 100,
    'penalty': 'l1',
    'solver': 'liblinear',
    'tol': 0.001
}

#Cross validation

n_fold=10
cv=skl_ms.KFold(n_splits=n_fold, random_state=2, shuffle=True)

for train_index, val_index in cv.split(x_training):
  X_train, X_val = x_training.iloc[train_index], x_training.iloc[val_index]
  y_train, y_val = y_training.iloc[train_index], y_training.iloc[val_index]

  model=skl_lm.LogisticRegression(**params)
  model.fit(X_train, y_train)
  prediction = model.predict(X_val)
  accuracy=np.mean(prediction==y_val)
  accuraccy_rates.append(accuracy)
  print('accuracy rate: ' + str(accuracy))

print('Mean accuracy rate: ' + str(np.mean(accuraccy_rates)))

# Define hyperparameters grid
param_grid = {
    'C': [0.1, 1.0, 10.0, 30.0],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga'],  # Different solvers
    'max_iter': [100, 200, 300],  # Different max_iter values
    'class_weight': [None, 'balanced'],  # Different class weight options
    'fit_intercept': [True, False],  # Whether to fit intercept
    'tol': [1e-3, 1e-4],  # Tolerance for stopping criteria
    'intercept_scaling': [1, 2]  # Scaling for intercept (relevant for
    ↪  'liblinear')
    }

# Initialize GridSearchCV
grid_search =
↪  GridSearchCV(estimator=skl_lm.LogisticRegression(solver='liblinear'),
                           param_grid=param_grid,
                            cv=cv)

# Fit the model
grid_search.fit(X_val, y_val)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Get the best model
best_LG = grid_search.best_estimator_
# Fit the best LG model
best_LG.fit(x_training, y_training)
## Split the data into training(80%) and validation sets(20%)
x_train, x_val, y_train, y_val = skl_ms.train_test_split(x_training,
↪  y_training, test_size=0.2, random_state=1)

# Implement and evaluate k-NN Classifier with k=2
knn = skl_nb.KNeighborsClassifier(n_neighbors=2).fit(x_train, y_train)
print("Test value: k = 2")
print(f"Accuracy: {accuracy_score(y_val, knn.predict(x_val))}\n")

# Experiment with different values of k to find the optimal one
misclassification = []
k_values = range(1, 50)  # 50 - 200 is optimal

for k in k_values:
    knn = skl_nb.KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train, y_train)
    y_pred = knn.predict(x_val)
    misclassification.append(np.mean(y_pred != y_val))

# Plotting misclassification rate vs. k
plt.plot(k_values, misclassification, marker='.')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Misclassification Rate')
plt.title('k-NN Varying number of Neighbors')
plt.show()

# Identify the optimal k value and retrain the model
optimal_k = k_values[misclassification.index(min(misclassification))]
print("For loop")
print(f"Optimal value of k: {optimal_k}")

# Retraining k-NN classifier with the optimal number of neighbors
knn_optimal = skl_nb.KNeighborsClassifier(n_neighbors=optimal_k)
knn_optimal.fit(x_train, y_train)
y_pred_optimal = knn_optimal.predict(x_val)
accuracy_optimal = accuracy_score(y_val, y_pred_optimal)
print(f"Optimized Accuracy with k = {optimal_k}: {accuracy_optimal}\n")

# Find the optimal k using GridSearchCV
```

```python
grid_search = GridSearchCV(skl_nb.KNeighborsClassifier(), {'n_neighbors':
↪  range(1, 50)}, cv=5, scoring='accuracy').fit(x_train, y_train)
print("Grid search")
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best model accuracy: {accuracy_score(y_val,
↪  grid_search.best_estimator_.predict(x_val))}\n")

# Find the optimal k using RandomizedSearchCV, 1 - 200 seems to be getting
↪  best result
random_search = skl_ms.RandomizedSearchCV(skl_nb.KNeighborsClassifier(),
↪  {'n_neighbors': range(1, 200)}, cv=5, scoring='accuracy',
↪  random_state=1).fit(x_train, y_train)
print("Random search")
print(f"Best parameters: {random_search.best_params_}")
print(f"Best model accuracy: {accuracy_score(y_val,
↪  random_search.best_estimator_.predict(x_val))}")
best_kNN = grid_search.best_estimator_

#training the best kNN model
best_kNN.fit(x_training, y_training)

# seperating the dataset into a training set and a validation set
x_train, x_val, y_train, y_val = skl_ms.train_test_split(x_training,
↪  y_training, train_size = 0.80, random_state = 1 )

#fitting the basic model
random_forest = RandomForestClassifier()
random_forest.fit(x_train, y_train)

# accuracy on training data
random_forest_score_train = random_forest.score(x_train, y_train) # 0.869
random_forest_score_val = random_forest.score(x_val, y_val)

print('Without tuning')
print(f'Random forest accuracy {random_forest_score_train} on training data')
print(f'Random forest accuracy {random_forest_score_val} on validation data')

# Hyperparameters to evaluate over (tuning)
param_grid = {
    'n_estimators': [25, 50, 100, 150],
    'max_features': ['sqrt', 'log2', None],
    'max_depth': [3, 6, 9],
    'max_leaf_nodes': [3, 6, 9],
}

grid_search = GridSearchCV(RandomForestClassifier(),
                           param_grid=param_grid)
grid_search.fit(x_val, y_val)
print(grid_search.best_estimator_)

# Training the best Random Forest model
best_RF = RandomForestClassifier(max_depth = 9, max_features = None,
↪  max_leaf_nodes = 6, n_estimators = 25)
best_RF.fit(x_training, y_training)
y_pred = best_RF.predict(x_val)
accuracy = accuracy_score(y_val, y_pred)
print(f' Accuracy of tuned random forest classifier {accuracy}')

# A dummy classifier for comparison of models.
model = DummyClassifier(strategy = 'uniform')
model.fit(x_training, y_training)
Dummy = model

#final test on hold out dataset
```

```python

models = []
models.append(Dummy)
models.append(best_LG)
models.append(best_LDA)
models.append(best_QDA)
models.append(best_kNN)
models.append(best_RF)
test_accuracy = []
test_fscore = []

for m in range(np.shape(models)[0]):
        model = models[m]
        pred = model.predict(x_testing)
        test_accuracy.append(np.mean(pred == y_testing))
        test_fscore.append(f1_score(y_testing, pred))


# Plotting test accuracy
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(('Dummy', 'LG', 'Lda', 'Qda', 'K-nn', 'Random Forest'), test_accuracy,
↪  marker='o', linestyle='-', color='skyblue')
plt.xlabel('Models')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy of Different Models')
plt.xticks(rotation=45)
plt.grid(True)

# Plotting F1 score
plt.subplot(1, 2, 2)
plt.plot(('Dummy', 'LG', 'Lda', 'Qda', 'K-nn', 'Random Forest'), test_fscore,
↪  marker='o', linestyle='-', color='salmon')
plt.xlabel('Models')
plt.ylabel('Test F1 Score')
plt.title('Test F1 Score of Different Models')
plt.xticks(rotation=45)
plt.grid(True)

plt.tight_layout()
plt.show()

print(f'Mean accuracy per model: {test_accuracy}')
print(f'Mean F score per model: {test_fscore}')

# to be able to download results
from google.colab import files


# training chosen model on the whole training dataset
best_RF.fit(x_final, y_final)

# loading the data
data = pd.read_csv('/content/test_data.csv', dtype={'ID':
↪  str}).dropna().reset_index(drop=True)

#do not modify this seed value, shuffle the train data in an another part of
↪  the code, so we all have the same test set

# Categorizing different variables
data['is_winter'] = data['month'].apply(lambda x: 1 if x in [12, 1, 2] else 0)
data['is_spring'] = data['month'].apply(lambda x: 1 if x in [3, 4, 5] else 0)
data['is_summer'] = data['month'].apply(lambda x: 1 if x in [6, 7, 8] else 0)
data['is_fall'] = data['month'].apply(lambda x: 1 if x in [9, 10, 11] else 0)
```

```
data['rush_hour'] = data['hour_of_day'].apply(lambda x: 1 if (7 <= x <= 9) or
↪  (17 <= x <= 19) else 0)
data['night_time'] = data['hour_of_day'].apply(lambda x: 1 if (20 <= x) or (7
↪  >= x) else 0)
data['is_there_snow'] = data['snowdepth'].apply(lambda x: 1 if (0 < x) else 0)

# dropping variables from the dataframe
columns_to_drop = ['snowdepth','summertime','day_of_week','month','dew',
↪  'precip', 'snow']
data.drop(columns=columns_to_drop, inplace=True)

#normalizing numerical coloumns
normalized_cols = ['temp', 'humidity', 'windspeed', 'cloudcover',
↪  'visibility']
dummy_cols = ['hour_of_day', 'holiday', 'weekday']

scaler = StandardScaler()
data[normalized_cols] = scaler.fit_transform(data[normalized_cols])

y_pred = best_RF.predict(data)
np.savetxt("predictions.csv", y_pred, delimiter=",")

files.download("predictions.csv")
```

A.2 Data visualization

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score
import sklearn.discriminant_analysis as skl_da
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 sklearn.preprocessing import StandardScaler
from sklearn.metrics import f1_score
import seaborn as sns

data = pd.read_csv('/content/training_data.csv', dtype={'ID':
↪  str}).dropna().reset_index(drop=True)

# Map categorical values to numerical values
data['increase_stock'] = data['increase_stock'].map({'low_bike_demand': 0,
↪  'high_bike_demand': 1})


data.shape
display(data)
data.info()
data.describe()

# Count occurrences of increase_stock=1 for each hour_of_day
hourly_counts_all = data[data['increase_stock'] ==
↪  1]['hour_of_day'].value_counts().sort_index()

# Create a Series containing counts of increase_stock=1 for all hours
↪  (including those with count 0)
all_hours_counts = hourly_counts_all.reindex(range(24), fill_value=0)

```

```python
# Create the bar plot
plt.figure(figsize=(12, 6))
sns.barplot(x=all_hours_counts.index, y=all_hours_counts.values)
plt.title('Count of increase_stock=1 per hour of the day (All hours)')
plt.xlabel('Hour of the Day')
plt.ylabel('Count where increase_stock=1')
plt.show()

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Drop rows with missing values in the 'day_of_week' column
increase_stock_1 = increase_stock_1.dropna(subset=['day_of_week'])

# Count occurrences of increase_stock=1 for each day_of_week
day_of_week_counts =
↪  increase_stock_1['day_of_week'].value_counts().sort_index()

# Create the bar plot
plt.figure(figsize=(12, 6))
sns.barplot(x=day_of_week_counts.index, y=day_of_week_counts.values)
plt.title('Count of high bike demand per day_of_week')
plt.xlabel('day_of_week')
plt.ylabel('Count where increase_stock=1')
plt.show()

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each temp
temp_counts = increase_stock_1['temp'].value_counts().sort_index()

# Find the range of temp values in the dataset
temp_range = range(int(data['temp'].min()), int(data['temp'].max()) + 1)

# Fill in missing temp counts with zeros
for temp in temp_range:
    if temp not in temp_counts.index:
        temp_counts[temp] = 0

# Sort the index after adding the missing values
temp_counts = temp_counts.sort_index()

# Create the bar plot
plt.figure(figsize=(60, 6))
sns.barplot(x=temp_counts.index, y=temp_counts.values)
plt.title('Count of high bike demand per temperature')
plt.xlabel('Temperature')
plt.ylabel('Count where increase_stock=1')
plt.show()

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each dew
dew_counts = increase_stock_1['dew'].value_counts().sort_index()

# Find the range of dew values in the dataset
dew_range = range(int(data['dew'].min()), int(data['dew'].max()) + 1)

# Fill in missing dew counts with zeros
for dew in dew_range:
    if dew not in dew_counts.index:
        dew_counts[dew] = 0
```

```python

# Sort the index after adding the missing values
dew_counts = dew_counts.sort_index()

# Create the bar plot
plt.figure(figsize=(60, 6))
sns.barplot(x=dew_counts.index, y=dew_counts.values)
plt.title('Count of high bike demand per dew')
plt.xlabel('Dew')
plt.ylabel('Count where increase_stock=1')
plt.show()

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each humidity
humidity_counts = increase_stock_1['humidity'].value_counts().sort_index()

# Find the range of humidity values in the dataset
humidity_range = range(int(data['humidity'].min()), int(data['humidity'].max())
↪  + 1)

# Fill in missing humidity counts with zeros
for humidity in humidity_range:
    if humidity not in humidity_counts.index:
        humidity_counts[humidity] = 0

# Sort the index after adding the missing values
humidity_counts = humidity_counts.sort_index()

# Create the bar plot
plt.figure(figsize=(200, 6))
sns.barplot(x=humidity_counts.index, y=humidity_counts.values)
plt.title('Count of high bike demand per humidity')
plt.xlabel('Humidity')
plt.ylabel('Count where increase_stock=1')
plt.show()

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each windspeed
windspeed_counts = increase_stock_1['windspeed'].value_counts().sort_index()

# Find the range of windspeed values in the dataset
windspeed_range = range(int(data['windspeed'].min()),
↪  int(data['windspeed'].max()) + 1)

# Fill in missing windspeed counts with zeros
for windspeed in windspeed_range:
    if windspeed not in windspeed_counts.index:
        windspeed_counts[windspeed] = 0

# Sort the index after adding the missing values
windspeed_counts = windspeed_counts.sort_index()

# Create the bar plot
plt.figure(figsize=(60, 6))
sns.barplot(x=windspeed_counts.index, y=windspeed_counts.values)
plt.title('Count of high bike demand per windspeed')
plt.xlabel('Windspeed')
plt.ylabel('Count where increase_stock=1')
plt.show()

```

```python
# Define the number of bins and create bins for visibility
num_bins = 10  # You can adjust the number of bins as needed
visibility_bins = pd.cut(data['visibility'], bins=num_bins)

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each visibility bin
visibility_counts =
↪  increase_stock_1.groupby(visibility_bins)['increase_stock'].count()

# Create the bar plot
plt.figure(figsize=(12, 6))
visibility_counts.plot(kind='bar')
plt.title('Count of increase_stock=1 for each visibility bin')
plt.xlabel('Visibility Bins')
plt.ylabel('Count where increase_stock=1')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.show()

# Define the number of bins and create bins for snowdepth
num_bins = 100  # You can adjust the number of bins as needed
snowdepth_bins = pd.cut(data['snowdepth'], bins=num_bins)

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each snowdepth bin
snowdepth_counts =
↪  increase_stock_1.groupby(snowdepth_bins)['increase_stock'].count()

# Create the bar plot
plt.figure(figsize=(60, 6))
snowdepth_counts.plot(kind='bar')
plt.title('Count of increase_stock=1 for each snowdepth bin')
plt.xlabel('Snowdepth Bins')
plt.ylabel('Count where increase_stock=1')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.show()

# Define the number of bins and create bins for cloudcover
num_bins = 30  # You can adjust the number of bins as needed
cloudcover_bins = pd.cut(data['cloudcover'], bins=num_bins)

# Filter the DataFrame for increase_stock=1
increase_stock_1 = data[data['increase_stock'] == 1]

# Count occurrences of increase_stock=1 for each cloudcover bin
cloudcover_counts =
↪  increase_stock_1.groupby(cloudcover_bins)['increase_stock'].count()

# Create the bar plot
plt.figure(figsize=(12, 6))
cloudcover_counts.plot(kind='bar')
plt.title('Count of increase_stock=1 for each cloudcover bin')
plt.xlabel('Cloudcover Bins')
plt.ylabel('Count where increase_stock=1')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.show()

!pip install ydata_profiling
!pip install lida==0.0.10 kaleido python-multipart uvicorn lmx==0.0.15a0
↪  tensorflow-probability==0.22.0

```

```python
from ydata_profiling import ProfileReport


profile = ProfileReport(data,title="Bike sharing data report")
profile.to_notebook_iframe()
profile.to_file("eda.html")
```