Do we need more bikes?
Project in Machine Learning

December 22, 2023
Statistical Machine Learning - 1RT700
Uppsala University

Authors:
Sigge Axelsson, Andreas Grönlund, Eskil Silfur and Kaj Törngren Sato

Abstract

The District Department of Transportation in Washington D.C. need to determine whether they should increase the availability of bicycles to the public. This project utilized machine learning methods to address the issue. The provided data set was analyzed to identify any underlying trends. Five models were trained on the data set and subsequently evaluated based on performance of different metrics. The models were logistic regression, LDA, QDA, K-nearest neighbor and Random Forest. The Random Forest model stood out with the best F1-score and accuracy, making it the chosen model for implementation in a 'in production' scenario.

1 Introduction

In the field of urban transportation management, the presence of public bicycle-sharing systems has become crucial for advancing sustainable and environmentally friendly commuting choices. As part of the Statistical Machine Learning course (1RT700), this project dives into the realm of classification to address a critical question faced by the District Department of Transportation in Washington D.C.: Should the number of bikes in the public bicycle-sharing system be increased during specific hours? The task at hand is framed as a binary classification challenge, with the goal of predicting whether the demand for bicycles will surge or remain stable based on multiple features. These features include date and time, weather conditions, temperature, and other more relevant features. The data set consists of 1600 observations chosen randomly from the Washington, D.C. city data over the last three years.

2 Data analysis

2.1 Analysis of feature type: Categorical or numerical?

The given data can be divided into numerical and categorical features. Numerical data has a natural ordering where one value is larger than the other, and you can perform arithmetic with it. It can be represented by discrete or continuous values. Categorical values, however, lack a natural ordering and are always discrete [1]. The features in the data set are described in Table 1.

Table 1: Division of numerical- and categorical features

Numerical	temp, dew, humidity, precip, snow, snow_depth, windspeed, cloudcover, visibility
Categorical	hour_of_day, days_of_week, month, holiday, weekday, summertime, increase_stock

2.2 Analysis of bike demand trends

To get a better understanding of the numerical features hour_of_day, day_of_week, and month, three kernel density estimate plots were made, seen in Figure 1. Analyzing the features, some trends could be found: For the feature hour of the day, we find that high bike demand is at its greatest between hours 14-19 with a peak around hours 16-17. Looking at the day of the week, we find a more flat distribution of high bike demand but with a peak around the weekend (days 4-6). Looking at the distribution of months, we see that high bike demand is mainly and evenly distributed between months 3 to 10, with low bike demand from month 10 to 3.

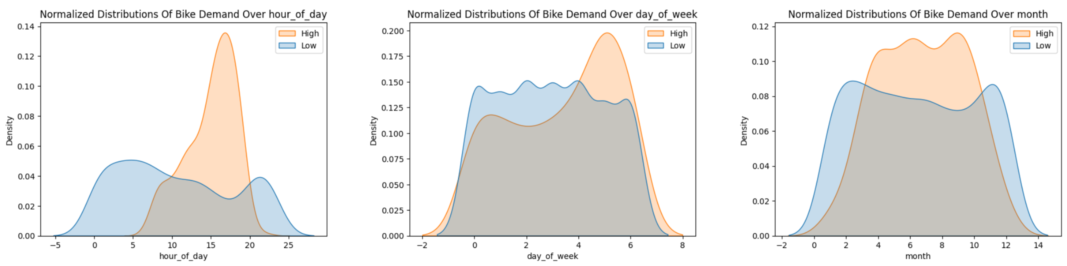


Figure 1: Normalized kernel density estimate plots of hour_of_day (left), day_of_week (middle), month (right)

Analyzing the categorical features; weekdays, holidays, and summertime, seen in Table 2, we can find trends in two of the three features. Firstly, analyzing holidays, no trend could be determined as the percentage of data points with high demand were very low compared to the non-holidays. Furthermore, it was found that there are only 53 data points of holidays, which could be too low to determine any statistics. Analyzing the weekdays, a trend could be found where high demand was more common during non-weekdays. Lastly, a clear trend in bike demand could be found during summertime, going from 6.84% high demand when it is not summer to 24.1% high bike demand during summertime.

Table 2: Trends during special days

	Holiday	Non-Holiday	Weekday	Non-Weekday	Summertime	Non-Summertime
Percentage	17.0%	18.0%	15.1%	25.0%	24.2%	6.84%
Samples	53	1547	1136	464	1030	570

Analyzing the weather, it was found that there was rarely high bike demand on days with rainfall and days where snow depth was not zero. During rainy days, high bike demand was most common on days where the precipitation was zero or under 0.2 mm. During snowy days, where snow depth was greater than zero, there were no days with high bike demand. The distribution of high bike demand can be found in Table 3.

Table 3: Rainy- and snowy days

	Rainy (> 0 mm precip.)	Non-Rainy	Snowy (> 0 mm depth)	Non-Snowy
Percentage	5.16%	19.4%	0.0%	18.7%
Samples	155	1445	58	1542

Other weather trends found were that high bike demand is more common around days with warmer temperatures with a peak around 20-30 ^{o}C and days where the humidity was around 40-50%. Looking at the wind speed, high bike demand seems to be mainly distributed between 10-20 km/h and has a mean skewed slightly to the right of its overall distribution. The distributions of bike demand over temperature, humidity, and wind speed can be found in Figure 2.

2.3 Outliers

To analyze outliers, a box plot (see Figure 4 in Appendix A) was made of the numerical features. The box plot indicated several outliers for a couple of features. Looking at the wind speed, three instances of high bike demand were found at wind speeds greater than 34 km/h, with one being at the high wind speed of 43.8 km/h. Looking at visibility, one instance of high bike demand was found when visibility was only 100 m. Looking at precipitation, the bike demand was always low during heavy rain, but notably, there was a data point with precipitation of 25.9 mm, which is more than double the nearest value.

We decided to remove 27 outliers from wind speed, which in turn also removed the greatest outlier in precipitation. We also chose to remove 5 outliers from visibility. The hope from removing these

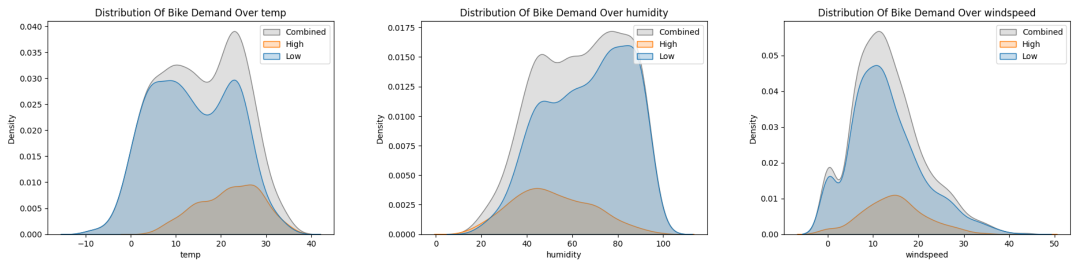


Figure 2: Kernel density estimate plots of temp (first, from left), humidity (second), cloudcover (third), windspeed (fourth)

outliers is that the data should better fit the average day while not removing too much information on days with bad weather.

2.4 Feature selection

To determine what features to use, the correlation with bike demand was analyzed. For categorical values, the chi-squared test was used, and for numerical features, the one-way ANOVA test was used. Both of these tests function by calculating a probability of features being not correlated; features with probabilities above 0.05 are considered to have no correlation. The results for all the features can be found in table A, figure 6. Noteworthy results were: Holiday had a score of 0.98 indicating no correlation to bike demand; furthermore, snow depth was found to have a correlation of 0.057, and visibility was found to have a correlation score of 0.068. Holiday was chosen to be removed, and snow depth was removed. The correlation results after outliers were removed can be found in A, figure 7.

Analyzing the correlation (using Pearson correlation this time, since both features are numerical) between the features, it was found that dew and temperature had a high correlation between each other of 0.87 but temperature having a greater ANOVA score towards bike demand; thus, we chose to remove the feature dew, correlation matrix of numerical features can be found in appendix A, fig 3.

Furthermore, it was found that the feature snow contained no information and was therefore removed.

2.5 Feature engineering

Snow depth was converted to a binary class 'snowy' for better interpretability by the algorithms. The categorical feature snowy was found to have better correlation to bike demand than snow depth.

3 Model development

In this project four families of classification methods have been studied and implemented. Each method describes its mathematical theory, how they are applied to the data and how well they perform. At the end a model selection is made where the best method is decided for a best 'in production' usage based on best performance of different metrics.

The training set was formed by randomly selecting 75% of the data, and the remaining 25% was designated for validation. The random number generator utilized a seed value of 1. Implementation of all methods can be seen in Appendix B, containing Python code for each method.

3.1 Naive model

A naive model was created that randomly chooses an output high- or low bike demand based on a statistical distribution modelled based on the provided data set. 82% of the labels in the training data were low bike demand and 12% were high bike demand. The naive models were modeled as random distribution to follow this pattern.

3.2 Logistic regression

Logistic regression is one method of modelling conditional class probabilities, predicting the probability of an instance belonging to a class. It is seen as a modification of linear regression in order to fit classification problems instead of regression problems. The linear regression model is described as:

$$f(x; \boldsymbol{\theta}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d = \boldsymbol{\theta}^T \boldsymbol{x} \tag{1}$$

where $\boldsymbol{x}$ is a vector containing the features and $\boldsymbol{\theta}$ the parameters. The logistic regression combines the linear regression model with the logistic function $\frac{e^x}{1+e^x}$, to model the class probability as (without noise term):

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

A high output is represented by $y = 1$ and a low output by $y = -1$. The linear regression model $f(x;\boldsymbol{\theta})$ takes on values across the entire real axis, but the fundamental idea behind logistic regression is to constrain these values to the interval [0, 1] through the use of the logistic function. Consequently, Equation 2 is restricted to [0, 1], allowing it to be interpreted as a probability. The model $\boldsymbol{\theta}$ is learned by minimizing the cost function (average loss):

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i; \boldsymbol{\theta}) \tag{3}$$

with the aim to reduce the expected new error

$$E_{\text{new}}(\boldsymbol{\theta}) = \mathbb{E}\left[E(f(x_*;\boldsymbol{\theta}), y_*)\right] \tag{4}$$

and $L(x_i, y_i; \boldsymbol{\theta})$, logistic loss, is given by

$$L(x_i, y_i; \boldsymbol{\theta}) = \ln[1 + e^{-y_i \boldsymbol{\theta}^T \boldsymbol{x}_i}] \tag{5}$$

[1] The logistic regression model was created using scikit-learn and the method L-BFGS was utilized for training. The model was fine-tuned by employing grid search on the hyperparameters, carried out over 10 folds. Cross validation was also made for evaluation.

Initially, the threshold value r was set to 0.5. In a binary classification, it is typically set to 0.5 where it convert predicted probabilities into class labels, where a value of 0.5 represents a balanced decision boundary. But after the tuning of parameters, the model acquired its highest accuracy of 86.5% where the threshold value was 0.41. This tuning aimed to find a more optimal threshold for the specific characteristics of the given data. A Precision-recall curve and Receiver Operating Characteristic (ROC) curve was also made to analyze and to visualize the model's performance across varying probability thresholds. They are shown in Figure 5 and Figure 6 in Appendix A.
The precision-recall curve is more informative for imbalanced problem where in this case the data set is majority belonging to one class, the negative class; low bike demand. The imbalance in data implies that a classifier that consistently predicts negative outcomes will exhibit a favorable performance in terms of the misclassification rate. A lower threshold classifies more instances as positive, increasing recall but potentially lowering precision, and vice versa.

3.3 Discriminant analysis: LDA, QDA

Linear Discriminant Analysis (LDA) is a classification and dimensionality reduction technique widely used in statistics and machine learning. It is a supervised method, meaning it requires labeled training data to learn the relationships between the features and the classes. LDA determines a decision boundary based on the linear combination of features. The decision boundary is chosen to maximize the separation between the means of different classes while minimizing the variance within each class.

Quadratic Discriminant Analysis (QDA) is a statistical classification technique that extends the principles of LDA to accommodate non-linear decision boundaries between classes. Unlike LDA, QDA does not assume equal covariance matrices for different classes, allowing it to capture more flexible and complex relationships in the data and enabling the model to handle cases where the amount of variation within each group is different. The decision boundaries produced by QDA are quadratic in nature, which makes it suitable for capturing non-linear relationships between features. QDA is particularly useful when dealing with multiclass classification problems and data sets with non-linear separations among classes. However, it may be sensitive to outliers and require larger sample sizes compared to linear discriminant methods.

3.3.1 Model

In order to reduce collinearity the QDA model was preprocessed with Principal Component Analysis (PCA), in order to reduce the dimensionailty and optimize the training data.

It is typically assumed that the class-conditional densities follow a Gaussian distribution. This means that the class conditional densities can be written as:

$$p(\boldsymbol{x}, y) = p(\boldsymbol{x} \mid y)p(y) \tag{6}$$

and

$$p(\boldsymbol{x}, y) = \frac{1}{\sqrt{(2\pi)^{\frac{d}{2}} \boldsymbol{\Sigma}^{\frac{d}{2}}}} exp(-\frac{1}{2}(x - \boldsymbol{\mu}_m)^T \boldsymbol{\Sigma}_m^{-1}(x - \boldsymbol{\mu}_m)) \tag{7}$$

$$p(\boldsymbol{x}, y) = p(\boldsymbol{x} \mid y = m; \boldsymbol{\theta}) = \pi_m \tag{8}$$

$\boldsymbol{\mu}_m$ is the mean value vector of a given class m and in LDA Σ_m is the covariance matrix shared by all classes, and π_m is the probability density function for a class $\pi = m$ and d is the dimensionality of the feature space. However, unlike LDA in QDA the classes does not share a common covariance matrix, the covariance matrix needs to be calculated for each class. It is calculated by Equation 9.

$$\Sigma_y = \frac{1}{N_y - 1} \sum_y i = y(x_i - \mu_y)(x_i - \mu_k)^T \tag{9}$$

The optimal parameters $\hat{\mu}_m$, $\hat{\Sigma}_m$ and $\hat{\pi}_m$ are defined by:

$$\hat{\mu}_m = \frac{1}{n_m} \sum_i x_i,\ \hat{\Sigma}_m = \frac{1}{n_m} \sum_i (\boldsymbol{x_i} - \boldsymbol{\hat{\mu}_m})(\boldsymbol{x_i} - \boldsymbol{\hat{\mu}_m})^T,\ \hat{\pi}_m = \frac{n_m}{n} \tag{10}$$

The visualization of data distribution involves considering the total number of training data points, n, and the specific number of training data points within a given class, n_y, This distribution could be represented through a contour plot, where the highest concentration of probability mass is visualized at the center of the contour. The probability of a prediction being accurate is greater closer to this central region and diminishes as the distance from the center increases.

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \text{argmax } p(\boldsymbol{x}|y = m)p(y = m) = \text{argmin}(\boldsymbol{x} - \boldsymbol{\mu_m})^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu_m}) + \ln|\Sigma_m| - 2\ln\pi_m \tag{11}$$

The model learns the distribution of each class (increase/decrease), from the training data. When tasked with making predictions using new input data, the model identifies the class y that maximizes the joint distribution as defined in Equation 11.

Neither QDA nor LDA have any hyperparameters to be tuned. Some regularization techniques could be used to stabilize the covariance matrix estimation when the sample size is small, his was not needed.

3.4 K-nearest neighbor

The K-nearest neighbor (KNN) method is a distance based method, using the euclidean distance between the test data and the training data to make predictions. The process involves calculating the distance between the test input and all training inputs, denoted as $||x_i - x_||$ for $i = 1, \ldots, n$. Subsequently, the data point x_j with the shortest distance to x is identified, and its output is used as the prediction, expressed as $y(x_*) = y(x_j)$ [1]. As the closest training data determines the classification, opting for an odd value for K is advantageous to ensure a consistent majority in the decision-making process.

To implement KNN, the initial step is to partition the data set into training and testing data. This division facilitates the evaluation of the trained model. The ratio of training to testing data should be chosen to optimize model training while ensuring a representative test set.

Upon inserting the test data, the model generates predictions, which are then used for result evaluation. Evaluation metrics such as F1-score and accuracy are commonly employed to assess the performance of the KNN method.

For the KNN method the parameter K is not a certainty, and can cause varying results depending on what it is chosen as. Two methods for determining K are the elbow method and hyperparameter study. The hyperparameter study is done by keeping all but one parameter constant and varying the last one to find the optimal value. When doing a hyperparameter study on K with a training data size of 75% the optimal K was determined as 37. In Figure 7, the choice of K was plotted against the accuracy score.

3.5 Tree-based methods: Classification Trees and Random Forests

3.5.1 Classification Trees

A decision tree is a machine learning algorithm that works by dividing an input space into multiple disjoint regions, with each region having a constant value used to predict an output $\hat{y}(\mathbf{x}_*)$ [1]. In the context of the project "Do we need more bikes?" the input space consists of the features in our data set (`hour_of_day`, `week_day`, `months`, etc.) and outputs are the prediction labels, low and high bike demand.

The output prediction can be formulated with Equation 12, where L is the number of regions and $\hat{y}_l$ is the predicted label for region R_l [1]. The classification tree decides the output label for a region R_l by majority vote, formulated in Equation 13.

$$\hat{y}(\mathbf{x}_*) = \sum_{l=1}^{L} \hat{y}_l \mathbb{1}\{\mathbf{x}_* \in R_l\} \qquad (12)$$

$$\hat{y}_l = \text{Majority Vote}\{y_i : \mathbf{x_i} \in R_l\} \qquad (13)$$

The goal of the machine learning algorithm will be to split the input space into regions R_l, according to Equation 14, in a way that gives the lowest possible error for future predictions. Considering p number of inputs x_j ($j \in 1, \ldots, p$), this will be done by finding the best combination of threshold values s in combination with inputs x_j to split the input space, minimizing the splitting criteria in Equation 15 [1], where n is the number of data points in each region and Q is the cost function for each region.

$$R_{\text{left}}(j, s) = \{\mathbf{x} : x_j < s\}, \; R_{\text{right}}(j, s) = \{\mathbf{x} : x_j \geq s\} \qquad (14)$$

$$\arg \min_{j,s} \; n_{\text{left}} Q_{\text{left}} + n_{\text{right}} Q_{\text{right}} \qquad (15)$$

The cost function used for the decision trees in this project is the Gini index described in Equation 16 [1], where l is the node number and m is the classifier label (low- and high bike demand).

$$Q_l = \sum_{m=1}^{M} \hat{\pi}_{lm}(1 - \hat{\pi}_{lm}) \qquad (16)$$

$$\hat{\pi}_{lm} = \frac{1}{n_l} \sum_{i:\mathbf{x_i} \in R_l} \qquad (17)$$

3.5.2 Bagging

Bagging is a method of reducing the variance created from over-fitting. From the original data set, you create new data sets containing randomly picked samples from the original data set. You then train decision trees on each data set and average their results through majority vote in the classification case.

3.5.3 Random Forest

Random Forest is an algorithm with the aim to reduce the model's variance by adding randomness during the construction of each tree forest. Instead of considering all possible input variables when splitting nodes, Random Forest randomly selects a subset of input variables for each tree in the forest [1].

While this randomness increases the variance of individual trees, it is thought that the reduction in correlation caused by introducing an aspect of randomness results in an overall decrease in the averaged prediction variance and reducing the model's prediction error [1]. Bagging can be used with the Random Forest as another tool to decrease the variance.

To optimize the model, a hyperparameter study was made using grid search with a five-fold stratified cross-validation. According to J. N. van Rijn et. al [2], the hyperparameters which most often have the greatest impact on the Random Forest model are the minimum samples per leaf and the maximum features to consider for each tree in the forest, thus they were chosen as hyperparameters. F1-macro (avarage F1-score of low and high demand) was chosen as the scoring method for the grid search to counteract the fact that the data set was heavily imbalanced. Results from the hyperparameter were to use max_featers = 6 and min_samples_leaf = 2 study can be found in Appendix A, Figure 8. Additionally bootstrapping and 10000 trees were used in the forest to minimise the variance. Since more trees does not increase risk of over-fitting, more trees were considered, but it was found that adding more trees did not yield any better accuracy when evaluating on the training data, point to the variance having converged to its lowest possible state.

3.6 Evaluation and model selection

In the assessment of all model performance, a k-fold cross-validation was made, selecting $k = 10$ and using its mean. Its results are seen Table 4. Metrics such as accuracy, precision, recall and F1-score was used to evaluate the performance of all models. Accuracy and F1-score were the main metrics to assess and determine the most optimal method. F1-score summarises the precision and recall and was deemed as extra important as the data set was highly imbalanced with 18% belonging to high bike demand. The F1-score is ranging from zero to one with higher values signifying better performance [1]. The F1-score and the accuracy were best while using the Random Forest method, see Table 5, therefore it was the clear preference for implementation.

Table 4: Mean Cross Validation score for the used methods

	Log. reg.	LDA	QDA	KNN	R.Forest
CV score	0.84	0.78	0.96	0.83	0.89

Table 5: Performance measurement for the used methods in the data set

	Naive model	Log. reg.	LDA	QDA	KNN	R. Forest
Accuracy	0.71	0.90	0.78	0.96	0.86	**0.97**
F1-score (High)	0.00	0.50	0.35	0.89	0.41	**0.90**
F1-score (Low)	0.83	0.92	0.87	**0.98**	0.92	**0.98**
Precision (High)	0.21	0.70	0.00	**1.00**	0.73	0.97
Precision (Low)	0.83	0.88	0.82	0.96	0.87	**0.97**
Recall (High)	0.22	0.39	0.00	0.80	0.29	**0.85**
Recall (Low)	0.83	0.96	**1.00**	**1.00**	0.98	0.99

4 Conclusion

Not all machine learning methods are universally applicable, as evident in predicting bike demand. What suits one task might not be suitable for another. However, each method used outperformed the naive model, indicating their ability to capture underlying trends in the data set. Careful selection of data is crucial—identifying trends and eliminating unnecessary inputs enhances model accuracy. Evaluation methods should be chosen wisely based on the characteristics of testing and training data. For instance, while KNN and logistic regression showed similar accuracy scores, their F1-scores differed significantly, emphasizing the need for evaluation beyond accuracy alone. The Random Forest model emerged as the best-performing model based on all evaluation metrics. The QDA model suffered from very high collinearity. The training data also contained a high number of parameters, and the suspiciously high accuracy could also be due to overfitting of the training data. Therefore the QDA-model could not be entirely trusted.

References

[1] Andreas Lindholm et al. *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press, 2022. URL: `https://smlbook.org`.

[2] Jan N. van Rijn and Frank Hutter. 'Hyperparameter Importance Across Datasets'. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*. KDD '18. ACM, July 2018. DOI: `10.1145/3219819.3220058`. URL: `http://dx.doi.org/10.1145/3219819.3220058`.

A Additional figures

Variable	**p-value**
chi2:	
`hour_of_day`	1.84×10^{-89}
`day_of_week`	1.50×10^{-4}
`month`	1.75×10^{-14}
`snowy`	5.39×10^{-4}
`holiday`	0.9884
`weekday`	4.52×10^{-6}
`increase_stock`	0.0
avonva:	
`temp`	8.78×10^{-44}
`dew`	1.00×10^{-7}
`humidity`	1.12×10^{-36}
`precip`	0.0177
`snowdepth`	0.0573
`windspeed`	1.20×10^{-4}
`cloudcover`	0.0686
`visibility`	5.39×10^{-6}

Table 6: Chi-squared and ANOVA p-values for the data sets features

Variable	**p-value**
chi2:	
`hour_of_day`	1.56×10^{-90}
`day_of_week`	1.71×10^{-4}
`month`	1.56×10^{-13}
`snowy`	6.56×10^{-4}
`holiday`	1.0
`weekday`	6.15×10^{-6}
`increase_stock`	0.0
avonva:	
`temp`	1.68×10^{-42}
`dew`	2.61×10^{-7}
`humidity`	7.45×10^{-38}
`precip`	0.0042
`snowdepth`	0.0664
`windspeed`	6.30×10^{-6}
`cloudcover`	0.1029
`visibility`	7.99×10^{-7}

Table 7: Chi-squared and ANOVA p-values for the data sets features when outliers were removed

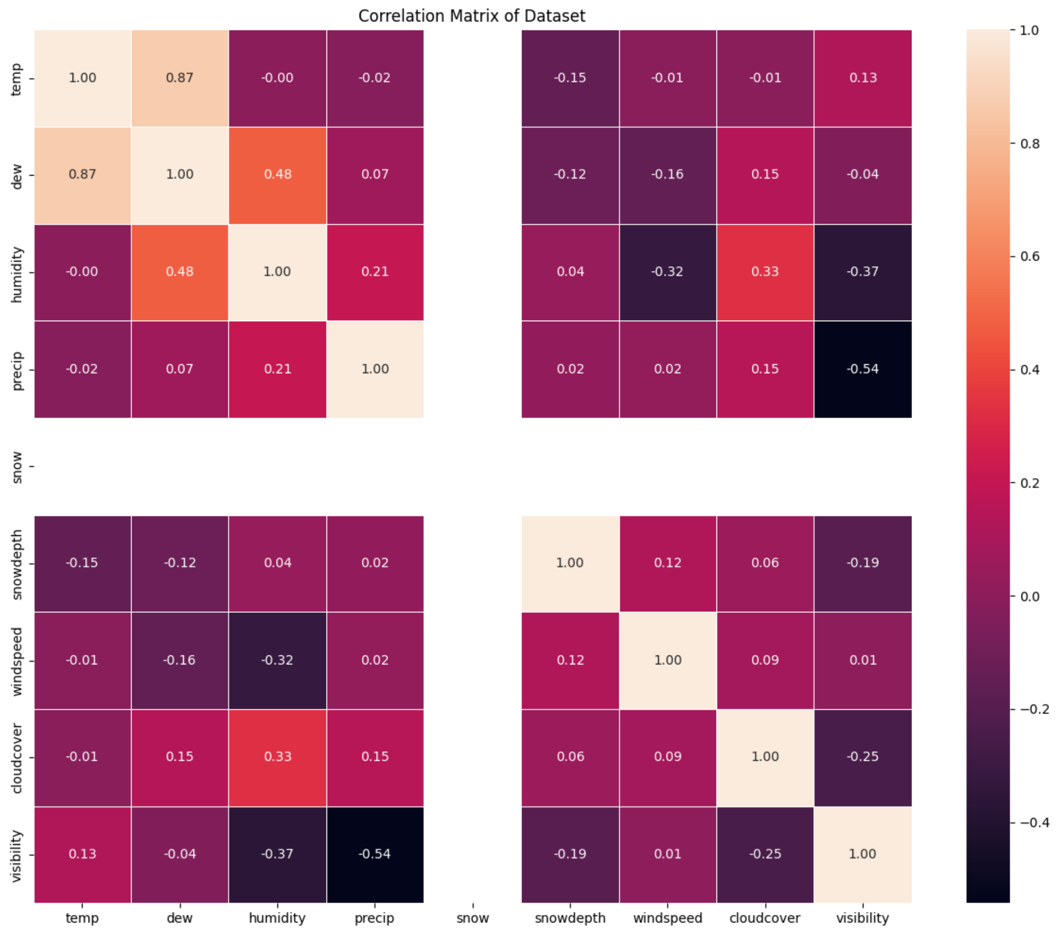


Figure 3: Correlation matrix of numerical features

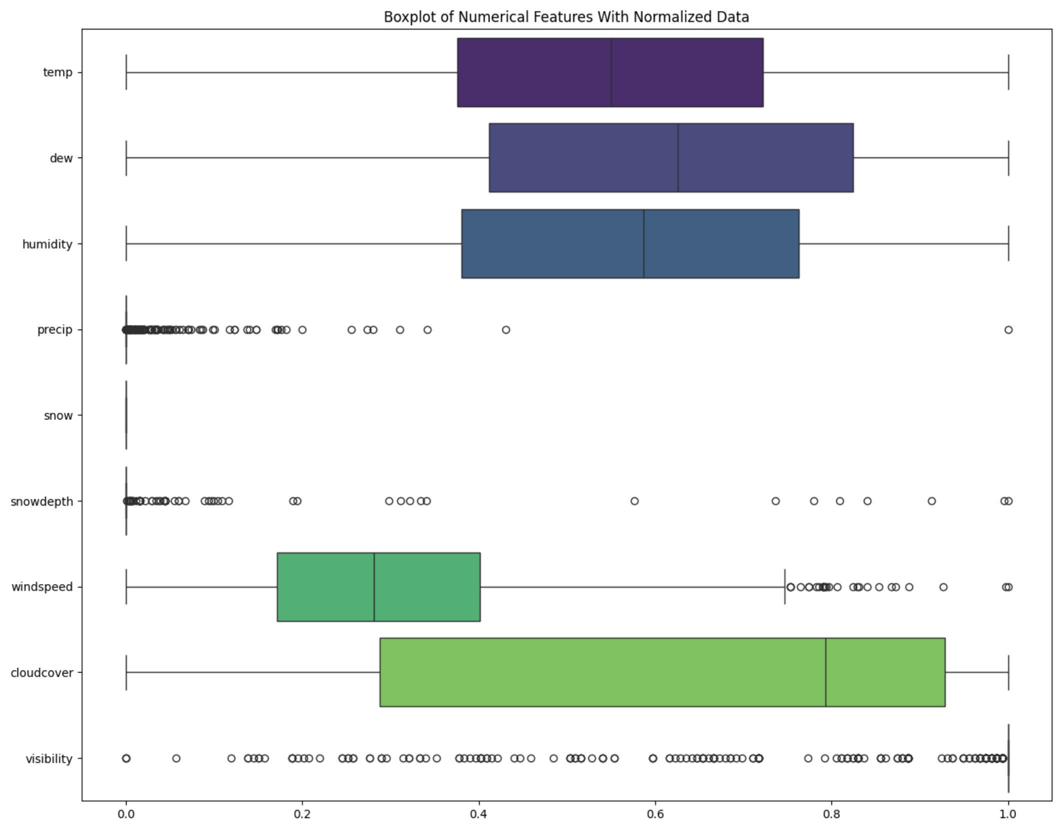


Figure 4: Boxplot of Numerical Features With Normalized Data

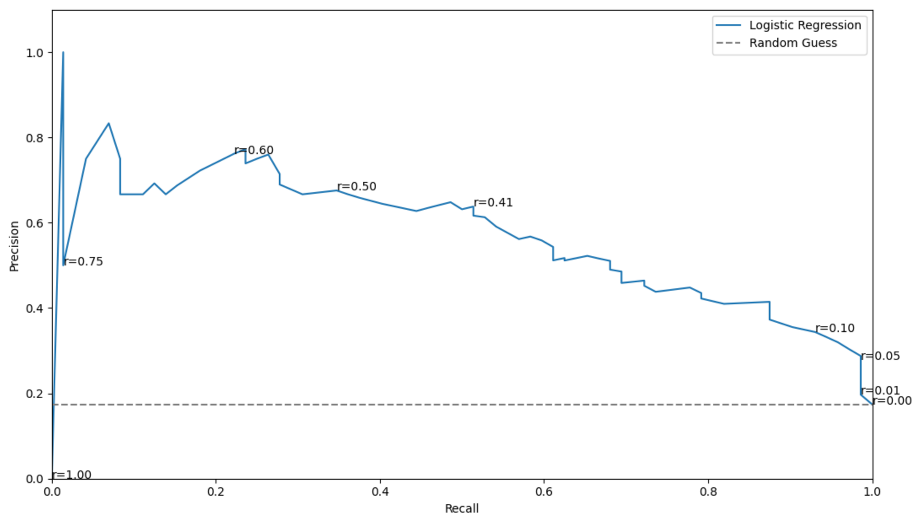


Figure 5: Precision-recall curve

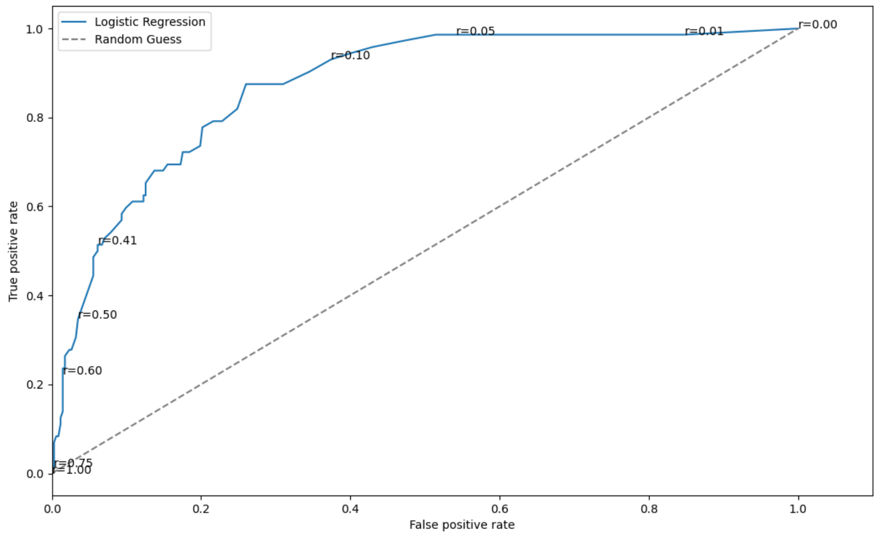


Figure 6: Receiver Operating Characteristic (ROC) curve

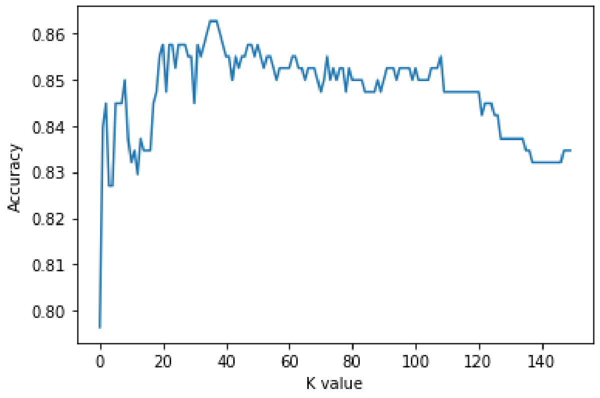
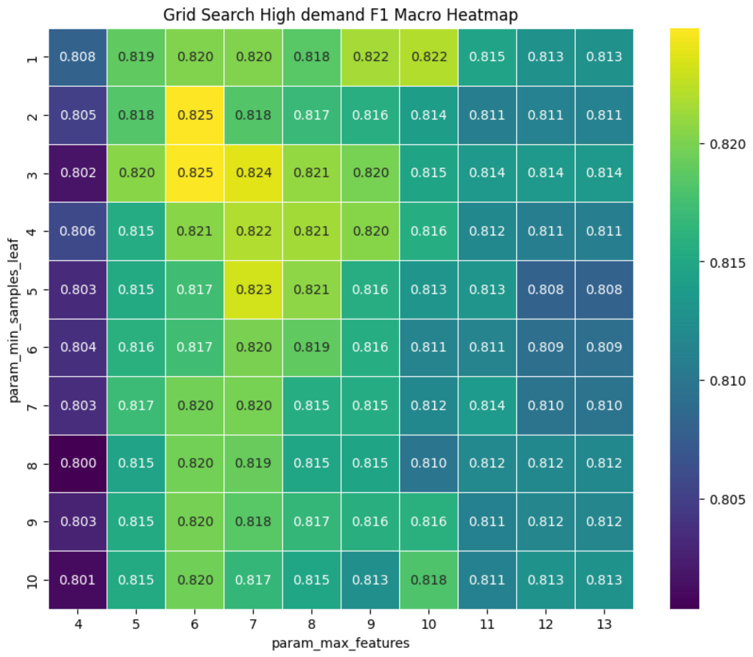


Figure 7: Hyperparameter study of K



param_min_samples_leaf \ param_max_features	4	5	6	7	8	9	10	11	12	13
1	0.808	0.819	0.820	0.820	0.818	0.822	0.822	0.815	0.813	0.813
2	0.805	0.818	0.825	0.818	0.817	0.816	0.814	0.811	0.811	0.811
3	0.802	0.820	0.825	0.824	0.821	0.820	0.815	0.814	0.814	0.814
4	0.806	0.815	0.821	0.822	0.821	0.820	0.816	0.812	0.811	0.811
5	0.803	0.815	0.817	0.823	0.821	0.816	0.813	0.813	0.808	0.808
6	0.804	0.816	0.817	0.820	0.819	0.816	0.811	0.811	0.809	0.809
7	0.803	0.817	0.820	0.820	0.815	0.815	0.812	0.814	0.810	0.810
8	0.800	0.815	0.820	0.819	0.815	0.815	0.810	0.812	0.812	0.812
9	0.803	0.815	0.820	0.818	0.817	0.816	0.816	0.811	0.812	0.812
10	0.801	0.815	0.820	0.817	0.815	0.813	0.818	0.811	0.813	0.813

Figure 8: Hyperparameter study of Random Forest

B Python code - Implementation

B.1 Data analysis code

```python
import pandas as pd
import matplotlib.pyplot as plt
from pandas.plotting import scatter_matrix

data = pd.read_csv("training_data.csv")

# Scatter plots for 'hour_of_day', 'day_of_week', 'month','holiday', '
    weekday', 'precip', 'snowdepth'  against 'increase_stock'
scatter_cols = ['hour_of_day', 'day_of_week', 'month', 'holiday', '
    weekday', 'precip', 'snowdepth', 'increase_stock']
scatter_data = data[scatter_cols]

# Color mapping for 'increase_stock'
color_map = {'low_bike_demand': 'yellow', 'high_bike_demand': 'red'}
scatter_data['color'] = scatter_data['increase_stock'].map(color_map)

# Scatter matrix
scatter_matrix(scatter_data, alpha=0.2, figsize=(12, 12), diagonal='
    hist', c=scatter_data['color'])

plt.show()
```

B.2 Logistic regression code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pandas.plotting import scatter_matrix
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.metrics as skmet

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

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


## Adding new features
rainy = []
for i in dataSet['precip']:
    if i > 0:
        rainy.append(1)
    else:
        rainy.append(0)
dataSet["rainy"] = rainy

snowy = []
for i in dataSet['snowdepth']:
    if i > 0:
        snowy.append(1)
    else:
        snowy.append(0)
dataSet["snowy"] = snowy

## Drop unecessary features
```

```python
dataSet = dataSet.drop(columns=['snow', 'precip', 'snowdepth', 'dew'])


## Outliers, windspeed

# Calculate the IQR (Interquartile Range)
Q1 = dataSet['windspeed'].quantile(0.25)
Q3 = dataSet['windspeed'].quantile(0.75)
IQR = Q3 - Q1

# Define the upper and lower bounds to filter out outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out rows with values outside the bounds
dataSet = dataSet[(dataSet['windspeed'] >= lower_bound) & (dataSet['windspeed'] <= upper_bound)]

# Split data into training and test sets
np.random.seed(1)

ratio = 0.75    # procent av dataset som ska vara tr ningsdata, resterande blir test
print(ratio*100, "% of testdata =", int(len(dataSet)*ratio))
trainIndex = np.random.choice(dataSet.shape[0], size=int(len(dataSet)*.75), replace=False) # V ljer slump ssigt ett antal index
trainIndexBool = dataSet.index.isin(trainIndex) # Ger en array med true/false baserat p   index finns i dataset
train = dataSet.iloc[trainIndex]                # V ljer tr ningsdata fr n dom platser i trainIndex som  r  True
test = dataSet.iloc[~trainIndexBool]            # Resterande blir testdata

X_train = train.copy().drop(columns=['increase_stock'])
Y_train = train['increase_stock']
X_test = test.copy().drop(columns=['increase_stock'])
Y_test = test['increase_stock']

model = skl_lm.LogisticRegression(solver='lbfgs', max_iter=1000)
model.fit(X_train, Y_train)
predict_prob = model.predict_proba(X_test)
print(model.classes_)
#predict_prob[0:5]

prediction = np.empty(len(X_test), dtype=object)
prediction = np.where(predict_prob[:, 0]>= 0.5, 'high_bike_demand', 'low_bike_demand')  #threshold r

# Confusion matrix
print("Confusion matrix:\n")
print(pd.crosstab(prediction, Y_test), '\n')

# Accuracy
print(f"Accuracy: {np.mean(prediction == Y_test):.3f}")

# Classification report
print("Classification Report:\n")
print(skmet.classification_report(Y_test, prediction))

# Error rate
pred = model.predict(X_test)
error = np.mean(pred != Y_test)
print('Error rate for logistic regression: ' +str(error))


```

```
# Grid search
from sklearn.model_selection import GridSearchCV
# Define the parameter grid for grid search
param_grid = [{
    'C': [0.001 , 0.01 , 0.1 , 1, 1.5 , 2, 5, 10],
    'penalty': ['l2'],
    'solver': ['lbfgs'],
},
{
    'C': [0.001 , 0.01 , 0.1 , 1, 1.5 , 2, 5, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear'],
},
]

# Perform grid
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy'
    )
grid_search.fit(X_train, Y_train)

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

# Train the model with the best hyperparameters
best_model = grid_search.best_estimator_
best_model.fit(X_train, Y_train)

# Make predictions on the test set
best_prediction = best_model.predict(X_test)

# Confusion matrix
print("\nAfter Grid Search:\n")
print("Confusion matrix:\n")
print(pd.crosstab(best_prediction , Y_test), '\n')

# Accuracy
print(f"Accuracy: {np.mean(best_prediction  == Y_test):.3f}")

# Classification report
print("Classification Report:\n")
print(skmet.classification_report(Y_test, best_prediction ))

# Error rate
error = np.mean(best_prediction  != Y_test)
print('Error rate for logistic regression:', error)


# Cross-validation
from sklearn.model_selection import cross_val_score

# Perform 10-fold cross-validation
cv_scores = cross_val_score(model, X_train, Y_train, cv=10, scoring='
    accuracy')
# Print cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean Cross-Validation Accuracy:", np.mean(cv_scores))



# ROC curve for logistic regression
true_positive_rate = []
false_positive_rate = []

positive_class = 'high_bike_demand'
```

```
negative_class = 'low_bike_demand'

P = np.sum(Y_test == positive_class)
N = np.sum(Y_test == negative_class)

threshhold = np.linspace(0.00, 1, 101)
positive_class_index = np.argwhere(model.classes_ == positive_class).
    squeeze()

for r in threshhold:
    prediction = np.where(predict_prob[:, positive_class_index] > r,
                          positive_class,
                          negative_class)

    FP = np.sum((prediction == positive_class) & (Y_test ==
    negative_class))
    TP = np.sum((prediction == positive_class) & (Y_test ==
    positive_class))

    false_positive_rate.append(FP/N)
    true_positive_rate.append(TP/P)

plt.plot(false_positive_rate, true_positive_rate, label='Logistic
    Regression')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Random
    Guess')
for idx in [0, 1 , 5, 10, 41, 50, 60, 75, 100]:
    plt.text(false_positive_rate[idx], true_positive_rate[idx], f"r={
    threshhold[idx]:.2f}")
plt.xlim([0,1])
plt.xlim([0,1.1])
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.legend()
plt.show()

# Precision-Recall curve for logistic regression
precision = []
recall = []

for r in threshhold:
    prediction = np.where(predict_prob[:, positive_class_index] > r,
                          positive_class,
                          negative_class)

    FP = np.sum((prediction == positive_class) & (Y_test ==
    negative_class))
    TP = np.sum((prediction == positive_class) & (Y_test ==
    positive_class))
    FN = np.sum((prediction == negative_class) & (Y_test ==
    positive_class))

    denominator = TP + FP
    if denominator == 0:
        precision.append(0.0)  # Set precision to zero if denominator
    is zero
    else:
        precision.append(TP / denominator)

    recall.append(TP / (TP + FN))

random_guess_precision = P / (P + N)
random_guess_recall = 1.0

plt.plot(recall, precision, label='Logistic Regression')
```

```python
plt.plot([0, 1], [random_guess_precision, random_guess_precision],
    linestyle='--', color='gray', label='Random Guess')
for idx in [0, 1 , 5, 10, 41, 50, 60, 75, 100]:
    plt.text(recall[idx], precision[idx], f"r={threshhold[idx]:.2f}")

plt.xlim([0, 1])
plt.ylim([0, 1.1])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()


# Find the optimal threshold for highest accuracy
best_threshold_accuracy = 0
best_accuracy = 0

for r in threshhold:
    prediction = np.where(predict_prob[:, positive_class_index] > r,
    positive_class, negative_class)

    # Calculate accuracy
    accuracy = np.mean(prediction == Y_test)

    # Update best threshold for accuracy
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_threshold_accuracy = r

print(f"Best Threshold for Highest Accuracy: {best_threshold_accuracy
    :.2f}")
print(f"Highest Accuracy: {best_accuracy:.3f}")
```

B.3 Discriminant analysis: LDA, QDA code

```python
import pandas as pd
import numpy as np
import sklearn.preprocessing as skl_pre
import sklearn.discriminant_analysis as skl_da
import sklearn.metrics as skmet
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

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

# Mapping Increase_Stock to have binary values
dataSet['increase_stock'] = dataSet['increase_stock'].map({'
    low_bike_demand': 0, 'high_bike_demand': 1})

# Adding new features
dataSet["rainy"] = np.where(dataSet['precip'] > 0, 1, 0)
dataSet["snowy"] = np.where(dataSet['snowdepth'] > 0, 1, 0)

# Drop unnecessary features
dataSet = dataSet.drop(columns=['snow', 'precip', 'snowdepth', 'dew'])

# Handle outliers in windspeed
Q1 = dataSet['windspeed'].quantile(0.25)
Q3 = dataSet['windspeed'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

```python
dataSet = dataSet[(dataSet['windspeed'] >= lower_bound) & (dataSet['
    windspeed'] <= upper_bound)]

# Split data into training and testing sets
data_train = np.random.seed(1)
trainI = np.random.choice(dataSet.shape[0], size=50, replace=False)
trainIndex = dataSet.index.isin(trainI)
train = dataSet.iloc[trainIndex]
test = dataSet.iloc[~trainIndex]
X_data_train = train
Y_data_train = train['increase_stock']
X_test = test
Y_test = test['increase_stock']


# Standardize the features (important for PCA)
scaler = skl_pre.StandardScaler()
X_train_standardized = scaler.fit_transform(train)
X_test_standardized = scaler.transform(test)

# Apply PCA to reduce collinearity
num_components = min(X_data_train.shape[0], X_data_train.shape[1])
pca = PCA(n_components=num_components)  # Keep all components for
    demonstration
X_data_train_pca = pca.fit_transform(X_train_standardized)
X_test_pca = pca.transform(X_test_standardized)

# Train a model on the reduced feature set (Quadratic Discriminant
    Analysis in this case)
model = skl_da.QuadraticDiscriminantAnalysis()
model.fit(X_data_train_pca, Y_data_train)

# Make predictions and evaluate the model
y_pred = model.predict(X_test_pca)
accuracy = accuracy_score(Y_test, y_pred)

print('Confusion matrix')
print(pd.crosstab(y_pred, Y_test))

print('Classification report\n')
print(skmet.classification_report(Y_test, y_pred))

# Perform 10-fold cross-validation
cv_scores = cross_val_score(model, X_data_train_pca, Y_data_train, cv
    =10, scoring='accuracy')
# Print cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean Cross-Validation Accuracy:", np.mean(cv_scores))

print(f'Accuracy: {accuracy:.3f}')
```

B.4 K-nearest neighbor code

```python
# -*- coding: utf-8 -*-
'KNN-Maskinl rning'
import numpy as np
import pandas as pd
import seaborn as sns


import sklearn.neighbors as skl_nb
import matplotlib.pyplot as plt
import sklearn.metrics as skmet

np.random.seed(1)
```

```

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

## Mapping Increase_Stock to have binary values
dataSet['increase_stock'] = dataSet['increase_stock'].map({'
    low_bike_demand': 0, 'high_bike_demand': 1})

## Adding new features
rainy = []
for i in dataSet['precip']:
    if i > 0:
        rainy.append(1)
    else:
        rainy.append(0)
dataSet["rainy"] = rainy

snowy = []
for i in dataSet['snowdepth']:
    if i > 0:
        snowy.append(1)
    else:
        snowy.append(0)
dataSet["snowy"] = snowy

## Drop unecessary features
dataSet = dataSet.drop(columns=['snow', 'precip', 'snowdepth', 'dew'])


## Outliers, windspeed

# Calculate the IQR (Interquartile Range)
Q1 = dataSet['windspeed'].quantile(0.25)
Q3 = dataSet['windspeed'].quantile(0.75)
IQR = Q3 - Q1

# Define the upper and lower bounds to filter out outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out rows with values outside the bounds
dataSet = dataSet[(dataSet['windspeed'] >= lower_bound) & (dataSet['
    windspeed'] <= upper_bound)]


trainI = np.random.choice(dataSet.shape[0], size=1200, replace=False)
trainIndex = dataSet.index.isin(trainI)
train = dataSet.iloc[trainIndex]
test = dataSet.iloc[~trainIndex]
X_train = train[['day_of_week','weekday', 'hour_of_day', 'month', '
    holiday', 'summertime', 'temp', 'humidity', 'windspeed', '
    cloudcover']]
Y_train = train['increase_stock']
X_test = test[['day_of_week','weekday', 'hour_of_day', 'month', '
    holiday', 'summertime', 'temp', 'humidity', 'windspeed', '
    cloudcover']]
Y_test = test['increase_stock']

acc = []
list_len = 150
model = skl_nb.KNeighborsClassifier(n_neighbors=57)
model.fit(X_train, Y_train)

#Hyperparameter
for k in range(list_len):
```

```python
    model = skl_nb.KNeighborsClassifier(n_neighbors=k+1)
    model.fit(X_train, Y_train)
    prediction = model.predict(X_test)
    acc.append(skmet.accuracy_score(Y_test, prediction))

K = acc.index(max(acc))+1
print(max(acc))
print(f'Optimal K: {K}')
plt.plot(range(list_len), acc)
plt.xlabel("K value")
plt.ylabel("Accuracy")

#model.fit (X_train, Y_train)
#prediction = model.predict(X_test)
print(pd.crosstab(prediction, Y_test), '\n')
print(skmet.classification_report(Y_test, prediction))
#print(skmet.accuracy_score(Y_test, prediction))
#F-measure
```

B.5 Random Fores

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn.metrics as skmet
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV, cross_val_score

## Import training data
dataSet = pd.read_csv('Projekt\\training_data.csv')

## Adding new features
snowy = []
for i in dataSet['snowdepth']:
    if i > 0:
        snowy.append(1)
    else:
        snowy.append(0)
dataSet["snowy"] = snowy

## Outliers, visibility
indexOutliers = dataSet[(dataSet['visibility'] <= 2)].index
print('Outliers:', len(dataSet[(dataSet['visibility'] <= 2)]))
dataSet = dataSet.drop(index=indexOutliers)

## Outliers, windspeed
indexOutliers = dataSet[(dataSet['windspeed'] >= 33)].index
print('Outliers:', len(dataSet[(dataSet['windspeed'] >= 33)]))
dataSet = dataSet.drop(labels=indexOutliers)

dataSet = dataSet.drop(columns=['snow', 'snowdepth', 'dew', 'holiday'
    ])

## Mapping Increase_Stock to have binary values
dataSet['increase_stock'] = dataSet['increase_stock'].map({'
    low_bike_demand': 0, 'high_bike_demand': 1})

# Move Increase_Stock to the rightmost column in the dataset
column_to_move = dataSet.pop('increase_stock')
dataSet['increase_stock'] = column_to_move

#### Splitting data
np.random.seed(1)
```

```

ratio = 0.75    # percentage of dataset to be training data, the rest
    becomes the test set
print(ratio*100, "% of test data =", int(len(dataSet)*ratio))
trainIndex = np.random.choice(dataSet.shape[0], size=int(len(dataSet)
    *.75), replace=False) # randomly choose a number of indices
trainIndexBool = dataSet.index.isin(trainIndex) # create an array with
     true/false based on whether the index is in the dataset
train = dataSet.iloc[trainIndex]                # select training data
     from the locations in trainIndex that are True
test = dataSet.iloc[~trainIndexBool]            # the remaining
    becomes test data

xTrain = train.copy().drop(columns=['increase_stock'])
yTrain = train['increase_stock']
xTest = test.copy().drop(columns=['increase_stock'])
yTest = test['increase_stock']

#### Hyperparameter Grid
n_estimators = [2000] # Number of trees in the random forest

max_features = [int(x) for x in np.linspace(4, 13, 10)] # Number of
    features to consider at every split

min_samples_leaf = [int(x) for x in np.linspace(1, 10, 10)] # Minimum
    number of samples required at each leaf node

bootstrap = [True] # Method of selecting samples for training each
    tree

# Create the hyperparameter grid
paramGrid = {'n_estimators': n_estimators,
            'max_features': max_features,
            'min_samples_leaf': min_samples_leaf,
            'bootstrap': bootstrap}

#### Grid Search
modelForrest = RandomForestClassifier(random_state=0)

search = GridSearchCV(estimator = modelForrest,
                        param_grid = paramGrid,
                        cv = 5,
                        verbose = 2,
                        n_jobs = -1,
                        return_train_score = True,
                        scoring='f1_macro')

search.fit(xTrain, yTrain)

# Create variables for the best model
bestModelForrest = search.best_estimator_
scores = search.cv_results_

#### Print best hyperparameters and evaluation
print('Best average CV score:', search.best_score_)
print('Best parameters:', search.best_params_)

data = {
    'param_max_features': list(search.cv_results_['param_max_features'
    ].data),
    'param_min_samples_leaf': list(search.cv_results_['
    param_min_samples_leaf'].data),
    'mean_test_score': list(search.cv_results_['mean_test_score'])
}
df = pd.DataFrame(data)
```

```

# Pivot the DataFrame
pivot_df = df.pivot(index='param_min_samples_leaf', columns='
    param_max_features', values='mean_test_score')

# Create a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(pivot_df, annot=True, cmap='viridis', fmt=".3f",
    linewidths=.5)
plt.title('Grid Search High demand F1 Macro Heatmap')
plt.show()

#### Train model
paramCopy = search.best_params_.copy()
paramCopy['n_estimators'] = 10000

modelForrest = RandomForestClassifier(**paramCopy, random_state=0,
    n_jobs=-1,)

modelForrest.fit(xTrain, yTrain)

# Print the best hyperparameters
yPredict = modelForrest.predict(xTest)
print(skmet.classification_report(y_true=yTest, y_pred=yPredict))
```

B.6 Naive model

```
import pandas as pd
import numpy as np
import sklearn.metrics as skmet


## Import training data
dataSet = pd.read_csv('Projekt\\training_data.csv')

## Adding new features
snowy = []
for i in dataSet['snowdepth']:
    if i > 0:
        snowy.append(1)
    else:
        snowy.append(0)
dataSet["snowy"] = snowy

## Outliers, visibility
indexOutliers = dataSet[(dataSet['visibility'] <= 2)].index
print('Outliers:', len(dataSet[(dataSet['visibility'] <= 2)]))
dataSet = dataSet.drop(index=indexOutliers)

## Outliers, windspeed
indexOutliers = dataSet[(dataSet['windspeed'] >= 33)].index
print('Outliers:', len(dataSet[(dataSet['windspeed'] >= 33)]))
dataSet = dataSet.drop(labels=indexOutliers)

dataSet = dataSet.drop(columns=['snow', 'snowdepth', 'dew', 'holiday'
    ])

## Mapping Increase_Stock to have binary values
dataSet['increase_stock'] = dataSet['increase_stock'].map({'
    low_bike_demand': 0, 'high_bike_demand': 1})

# Move Increase_Stock to the rightmost column in the dataset
column_to_move = dataSet.pop('increase_stock')
dataSet['increase_stock'] = column_to_move

```

```

#### Splitting data
np.random.seed(1)

ratio = 0.75    # percentage of dataset to be training data, the rest
    becomes the test set
print(ratio*100, "% of test data =", int(len(dataSet)*ratio))
trainIndex = np.random.choice(dataSet.shape[0], size=int(len(dataSet)
    *.75), replace=False) # randomly choose a number of indices
trainIndexBool = dataSet.index.isin(trainIndex) # create an array with
     true/false based on whether the index is in the dataset
train = dataSet.iloc[trainIndex]                # select training data
     from the locations in trainIndex that are True
test = dataSet.iloc[~trainIndexBool]            # the remaining
    becomes test data

xTrain = train.copy().drop(columns=['increase_stock'])
yTrain = train['increase_stock']
xTest = test.copy().drop(columns=['increase_stock'])
yTest = test['increase_stock']


#### Naive model
prob = len(dataSet[dataSet['increase_stock'] == 0])/len(dataSet['
    increase_stock'])
yNaiveDistributed = np.random.choice([0, 1], size=len(xTest), p=[prob,
     1-prob])

print('Naive Distributed:\n', skmet.classification_report(y_true=yTest
    , y_pred=yNaiveDistributed))
```