Do we need more bikes?
Project in Machine Learning

Wenhan Zhou
Department of Information Technology
Uppsala University

Chenglong Li
Department of Information Technology
Uppsala University

Yizhi Liu
Department of Information Technology
Uppsala University

Haote Liu
Department of Mathematics
Uppsala University

Abstract

A binary classification problem of bike demand tendency is the focus of this paper, which compares different machine learning methods for solving it. These methods are discriminant analysis, nearest neighbors, tree-based methods, and deep neural networks. The paper uses accuracy, precision, and recall to measure the performance of these methods. The results indicate that the deep neural network has the best performance on this tendency prediction task. There are 4 people in this group.

1 Introduction

The bike demand in Washington is affected by several factors, such as weather conditions and time of the day, among others. Accurate and reliable predictions of the bike demand are a crucial step in distributing resources more efficiently.

In this study, we have been given a tabular dataset with 1600 samples of 15 features, that may be related to low and high bike demand, see table 1. In addition to the 15 input features in the dataset, we are also given the corresponding labels, zeros, and ones, that correspond to low and high bike demand respectively. The problem is therefore formulated as a binary classification problem.

Our goal is to compare several machine-learning models that are capable of classifying the conditions where low or high demand occurs. The comparison involves Linear Discriminant Analysis, Quadratic Discriminant Analysis, k-nearest neighbors, tree-based methods such as the Random Forest Classifier, Boosting methods like Adaboost and XGBoost, and finally, an ensemble of Deep Neural Networks.

2 Data Analysis

In our data analysis, key findings emerged regarding the impact of time, weather, and day type on shared bicycle demand. Numerical and certain dual-character (numerical and categorical) features, detailed in table 1, are pivotal in understanding these trends.

Seasonal patterns in Washington D.C. show higher bicycle demand during warmer months and daytime hours, with a noticeable decrease during colder months like January- February and November-December. Weekdays experience higher demand compared to weekends, likely due to different travel habits and choices, with weekends being more leisure-oriented.

Weather conditions also significantly influence bicycle usage. Demand drops on colder or wetter days, indicating a direct correlation between less favorable weather and decreased bicycle sharing.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

In summary, our analysis reveals that time, weather, and day type are crucial determinants of shared bicycle usage patterns.

We scale each of the features with the min-max scaling for our dataset:

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}, \tag{1}$$

our motivation is that this removes the units of the features of the dataset, making them comparable to our machine-learning models.

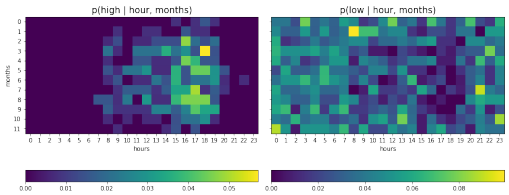

Figure 1: Given the hour of the day and the month of the year, we plot the probability of observing high and low bike demand respectively.

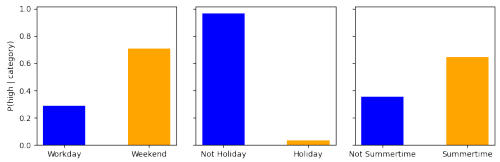

Figure 2: Conditional probability of high bike demand given some categorical feature. Note that there are many more cases where the days are not holidays.

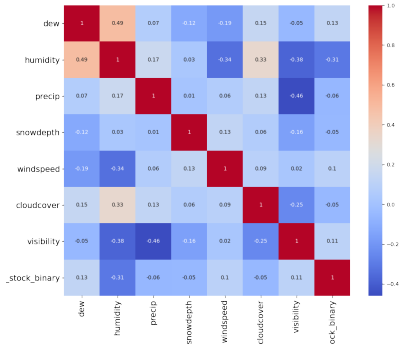

Figure 3: Correlation between weather features and bike demand.

Table 1: Data Type Table

Data Type	**Features**
Numerical Data	temp, dew, humidity, precip, snow, snowdepth, windspeed, cloudcover, visibility
Categorical Data	hour of day, day of week, month, holiday, weekday, summertime

3 Discriminant Analysis

Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) are statistical approaches used in machine learning for classifying a set of observations into different classes. These methods project input data onto a linear subspace, which maximizes the separation between classes (1). In LDA the decision surface is linear, while in QDA, it has the quadratic decision surfaces. The choice between LDA and QDA depends on the underlying distribution of the data and the complexity of the classification problem.

3.1 Mathematical Formulation

Both LDA and QDA model(1) the class conditional distribution $P(y|x)$ as a multivariate Gaussian distribution.

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} = \frac{P(x|y)P(y)}{\sum_i P(x|y_i)P(y_i)} \tag{2}$$

$$P(x|y) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y)\right), \tag{3}$$

Where n is the number of features, μ_y is the mean vector of the features that belong to class y, and Σ is the covariance matrix of the Gaussian for class y respectively.

QDA Log-posterior of QDA is given by:

$$\begin{aligned} \log P(y|x) &= \log P(x|y=k) + \log P(y=k) + C \\ &= -\frac{1}{2}\log|\Sigma_y| - \frac{1}{2}(x-\mu_y)^T\Sigma_y^{-1}(x-\mu_y) + \log P(y) + C, \end{aligned} \tag{4}$$

where C is a constant that is related to $P(x)$ and all other terms in the log-posteriro that do not depend on the class y. The outcomes predicted class from QDA is the one that maximizes this log-posterior.

LDA LDA is simply a special case of QDA with the assumption that covariance $Sigma$ is the same for all classes. Hence the log-posterior for LDA simplifies to:

$$\log P(y|x) = -\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y) + \log P(y) + C. \tag{5}$$

Since covariance $Sigma$ are the same for all classes, the LDA method can be seen as finding the class with mean vector that is "closest" to a given data point, with "closeness" measured by the Mahalanobis distance.

4 Nearest Neighbors

Nearest Neighbors(2) is one of the simplest machine learning algorithms that can be used for both classification and regression. It is a supervised learning algorithm. Upon analysis, it is determined that the problem is a classification task, for which the k-Nearest Neighbors (k-NN) algorithm is aptly suited. Employing grid search and 5-fold cross-validation, we optimize the model, selecting k=13 as the ideal parameter. This optimized model achieves an accuracy of 86%(see table 2 for details), demonstrating the effectiveness of k-NN in classification scenarios and the importance of targeted parameter tuning.

4.1 k-Nearest Neighbor

In k-NN, the prediction of a sample's label is based on the labels of its K nearest neighbors. The term "nearest" is usually determined based on a distance metric, such as the Euclidean distance.

We define the training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, test input $\mathbf{x}_*$ and the set $\mathcal{N}_* = \{i\}$, $\mathbf{x}_i$ is one of the k training data points closest to $\mathbf{x}_*$. The weight function:

$$\omega(\mathbf{x}, \mathbf{x}_i; k) = \begin{cases} 1, & \text{if } \mathbf{x}_i \text{ is } k \text{ nearest neighbour to } \mathbf{x} \\ 0, & \text{else} \end{cases} \tag{6}$$

A simple classifier based on the k-Nearest Neighbors (k-NN) algorithm can be described by the following general formula:

$$f(\mathbf{x}; k, \mathcal{T}) = \arg\max_y \sum_{i=1}^{n} \omega(\mathbf{x}, \mathbf{x}_i; k)\mathbb{I}_{\{y_i=y\}} \tag{7}$$

Depending on the purpose, k-NN prediction can be divided into regression and classification problems.

$$\hat{y}(\mathbf{x}_*) = \begin{cases} \text{Average}\{y_j : j \in \mathcal{N}_*\} & \text{(Regression problems)} \\ \text{Majority Vote}\{y_j : j \in \mathcal{N}_*\} & \text{(Classification problems)} \end{cases} \tag{8}$$

5 Tree-based Methods

5.1 Decision Trees

Decision trees are interpretable, simple machine learning algorithms effective for both categorical and numerical data. Their efficiency with tabular datasets enables the construction of powerful ensemble models using Bagging, see section 5.2 and Boosting, see section 6. The DT functions by creating a tree-like structure, consisting of root nodes (RNs), internal decision nodes (DNs), and leaf nodes (LNs). The DT starts from the RN and finds a decision boundary that minimizes a certain criterion. We choose to minimize the entropy:

$$S = -\sum_{a \in \mathcal{A}} \pi(y|a) \log \pi(y|a). \tag{9}$$

In this formula, $\mathcal{A}$ represents the space of possible split actions, and $\pi(y|a)$ represents the proportion of samples belonging to a class y in a DN, relative to the total number of samples at that node, given the split action a.

The action of reducing the entropy can be interpreted as reducing the disorder $\pi(y|a)$ creates when a node split has occurred. Given the optimal split, the algorithm follows the path to the next node. This branching continues until a leaf node is reached, which signifies the final decision or classification outcome.

This approach aims to create subsets of data that are as homogeneous as possible, thereby enhancing the clarity and accuracy of decisions made by the tree. The decision-making process is iterative and may stop when the maximum depth of the tree is reached. We provide a diagram visualizing DT on our dataset shown in figure 6. One of the most important hyper-parameters that controls the trade-off between bias and variance is the maximal tree depth. A deep DT will in general also contain many DNs. A complex DT may thus have low bias but high variance.

5.2 Bagging

Bagging stands for *Bootstrap AGGregatING*, an ensemble method used in machine learning. The term bootstrapping refers to sampling with replacement, in the context of our work, the training dataset is sampled such that each subset is independently identically distributed (IID). Furthermore, since we sample with replacement, each of the subsets likely has some overlapping elements.

With the new bootstrapped dataset, we may train a set of independent classifiers for each of the subsets of training data. Each of the classifiers is thus weakly correlated. After the completion, an aggregation step is performed. One type of aggregation operation, hard voting, works by counting the predicted label of each classifier, and the final prediction is made by picking the most frequent prediction. Another type of aggregation operation, soft voting, is performing an average of the probability of each classifier.

The Random Forest Classifier (RFC) (3) is a type of ensemble algorithm that uses bagging. Specifically, the RFC fits a set of DTs that is trained on a bootstrapped dataset, both in terms of samples, but also features. Each DT in a RFC is thus made as uncorrelated as possible. As a result, the combined effect of each DT contributes to reduced variance without increasing the bias.

5.3 Soft-voting ensemble

A general technique that we deploy in multiple models that is capable of predicting probability distribution is retraining the same model on the same training set using bagging, where each of the models is initialized with s different seeds:

$$H(x) = \frac{1}{s \cdot N_{\text{cv}}} \sum_{t=1}^{s} \sum_{j=1}^{N_{\text{cv}}} h_t(x; x_{\text{cv},j}), \tag{10}$$

where $x_{\text{cv},j}$ is the inputs corresponding to the j:th cross validation fold, h_t is a single model that is trained on $x_{\text{cv},j}$, $H(x)$ is the ensemble model, and $N_{\text{cv}} = 5$ is the number of cross validation folds.

For RFC in 2, the performance is measured for $s = 50$, e.g., an ensemble of $5 \cdot 50 = 250$ RFCs.

6 Boosting Methods

Bike-sharing usage depends on numerous factors that interact in a nonlinear fashion across different time horizons. Individual variables may carry limited signals on their own. Boosting overcomes this by allowing many simple base models, each capturing one aspect, to collectively build a sophisticated overall representation. The best hyperparameters for AdaBoost found via grid search were: learning_rate = 0.1, max_depth = 7, n_estimators = 100, while the best parameters for XGBoost were: learning_rate = 0.2, n_estimators = 50.

6.1 AdaBoost Algorithm

AdaBoost (Adaptive Boosting) is an ensemble learning method that combines weak learners to create a strong classifier. It was introduced by Yoav Freund and Robert Schapire in 1996(4). The basic idea behind AdaBoost is to focus on the mistakes made by weak learners and give more weight to misclassified samples, allowing subsequent weak learners to correct these mistakes.

The following is the basic algorithm flow of Adaboost. The description of mathematical symbols is in Appendix 3:
Input: Training dataset $D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, where x_i is the feature vector and y_i is the label; Weak learner algorithm; Number of iterations T.
Output: Ensemble strong classifier $H(x)$.

1. **Initialize Weights:** Assign equal weights to each sample, i.e., $w_i = \frac{1}{N}$.
2. **Iterative Training:**
   (a) For each round $t = 1, 2, ..., T$:
      i. Select the weak classifier with the lowest current error rate as the t-th base classifier h_t.
      ii. Compute the classifier h_t error rate and compute the weight for classifier h_t:

$$\varepsilon_t = \sum_{i=1}^{N} w_i^{(t)} \cdot \mathrm{I}(h_t(x_i) \neq y_i), \alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right). \tag{11}$$

      iii. Update sample weights, where Z_t is the normalization factor:

$$w_i^{(t+1)} = \frac{w_i^{(t)} \cdot \exp(-\alpha_t \cdot y_i \cdot h_t(x_i))}{Z_t}, Z_t = \sum_{i=1}^{N} w_i^{(t)} \cdot \exp(-\alpha_t \cdot y_i \cdot h_t(x_i)). \tag{12}$$

3. **Build Strong Classifier:**
   Construct the final strong classifier: $H(x) = \frac{1}{2} \left(\text{sign} \left(\sum_{t=1}^{T} \alpha_t \cdot h_t(x) \right) + 1 \right)$.

6.2 XGBoost Algorithm

XGBoost(Extreme Gradient Boosting) is a gradient-boosting algorithm designed for regression and classification problems. Proposed by Tianqi Chen in 2016 (5).

XGBoost is an ensemble learning algorithm based on decision tree models. It employs the gradient boosting approach, iteratively training a series of weak learners. Each iteration adjusts the data based on the previous round's predictions to minimize the prediction error.

Combining the loss function and regularization term, the XGBoost objective function is represented below, the description of mathematical symbols and the derivation of the equation are shown in Appendix 4:

$$\text{Obj} = \sum_{i=1}^{n} \frac{1}{2} (y_i - \hat{y}_i)^2 + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{13}$$

The goal is to minimize this objective function. XGBoost utilizes optimization algorithms such as gradient descent to achieve this objective.

Decision Tree Structure

XGBoost uses CART (Classification and Regression Trees) as the base learner. Each decision tree consists of nodes and leaves. Each node has a conditional statement dividing the data to left or right child.

Training Process of Weak Learners

In each iteration, XGBoost calculates residuals, representing the difference between the true values and the current predictions. A new tree is fit to the residuals to correct the model's errors. Model parameters are updated through gradient descent to minimize the objective function.

7 Deep Neural Network

We choose to implement a Deep Neural Network (DNN) for predicting bike demand due to its ability to learn complex, non-linear feature interactions, which allows it to produce a non-linear decision boundary. Furthermore, DNNs handle high-dimensional data efficiently, a key advantage when limited feature engineering is performed.

7.1 Logistic Regression

Generalized Linear Models (GLMs) extend traditional linear regression for an arbitrary number of parameters $\theta \in \Theta$

$$z = z(x;\theta) = x^T\theta. \tag{14}$$

We can modify the GLM to perform a binary classification task by applying the logistic function

$$\mathbb{P}(y|x;\theta) = \frac{1}{1+e^{-z}}. \tag{15}$$

The optimal parameters θ^* are estimated using Maximum Likelihood Estimation (MLE), which involves numerically minimizing the expected negative log-likelihood function

$$\theta^* = \arg\min_{\theta\in\Theta} \mathbb{E}\left[-\log \mathbb{P}(y|x;\theta)\right]. \tag{16}$$

The empirical expectation value above can be written as the binary cross-entropy (BCE) loss

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \mathbb{P}(y_i|x_i;\theta) + (1-y_i)\log \mathbb{P}(1-y_i|x_i;\theta). \tag{17}$$

The method of minimizing the BCE to find the optimal parameters given a GLM is known as logistic regression and is capable of producing a linear decision boundary that separates the label classes.

7.2 Non-linear Classification

The logistic regression is limited to producing linear decision boundaries by the GLM. Therefore, for a more complex dataset, we consider a natural non-linear generalization of the logistic regression, namely, Deep Neural Network (DNN). Instead of the GLM in 14, we consider a non-linear model with a weight matrix $\mathbf{W}$ and a offset vector $\mathbf{b}$ in the form

$$x_{l+1} = \sigma(x_l^T\mathbf{W}_l + \mathbf{b}_l), \tag{18}$$

where $1 \leq l \leq L$ denotes the number of layers of the DNN, $\mathbf{W}_l \in \mathbb{R}^{h\times h}$, $\mathbf{b}_l \in \mathbb{R}^h$, with h being the number of hidden units, and $\sigma(\cdot)$ represents a non-linear activation function. Note that we have:

$$x_0^T = x^T\mathbf{W}_{\text{in}} + \mathbf{b}_{\text{in}}, \tag{19}$$

where $\mathbf{W}_{\text{in}} \in \mathbb{R}^{\#\text{input features}\times h}$ and $\mathbf{b}_{\text{in}} \in \mathbb{R}^h$.

We use the Rectified Linear Unit (RELU) activation function, defined as

$$\sigma(\cdot) = \max(0,\cdot). \tag{20}$$

Since the relation between x_{l+1} and x_l is non-linear, we can construct an arbitrary complex regression model by considering some large value of L. Furthermore, we consider additional techniques such as residual connections (6)

$$x_{l+1} = \sigma(x_l^T \mathbf{W}_l + \mathbf{b}_l) + x_l, \tag{21}$$

to improve the training process. In addition, we also perform batch normalization (7), defined as

$$\mathrm{BN}_{\gamma_l,\beta_l}(x_l) = \gamma_l \cdot \frac{x_l - \mathbb{E}[x_l]}{\sqrt{\mathrm{Var}[x_l] + \varepsilon}} + \beta_l \tag{22}$$

with some trainable parameter γ_l and β_l and $\varepsilon > 0$ for numerical stability, before applying the non-linear function. The final non-linear regression model is thus

$$x_{l+1} = \sigma\left(\mathrm{BN}_{\gamma_l,\beta_l}(x_l^T \mathbf{W}_l + \mathbf{b}_l)\right) + x_l. \tag{23}$$

Since our goal is to perform non-linear binary classification, we can replace the GLM with the DNN such that

$$z(x_{l+1}; \theta) = x_{l+1}^T \theta + b, \tag{24}$$

where $\theta \in \mathbb{R}^{h \times 1}$ and $b \in \mathbb{R}$. Finally, the logistic function in equation 15 is applied to get the probability of predicting y, conditioned on x. The DNN in our case can thus be interpreted as a non-linear generalization of the logistic regression that is capable of producing non-linear decision boundaries.

The performance presented in table 2 is based on the same technique discussed in 5.3 with $s = 10$. We also over-parameterize the DNNs with $h = 200$, $L = 2$, trained with l_2-regularization, this method is inspired by the deep double descent phenomena (8). Figure 5 demonstrates that the ensemble is capable of achieving higher accuracy than the individual model.

8 Evaluation

To evaluate our models, we split the shuffled dataset into two parts, a larger part consisting of 75% of the original dataset. This part performs a 5-fold stratified cross-validation to find the desired hyperparameters. The models are then evaluated on the hold-out dataset with the remaining 25%. The results are shown in table 2.

The data analysis suggests that there are 82% of low bike demands (label 0). Our baseline model is thus only predicting 0, independent from the inputs.

Table 2: Comparison of model performance with low bike demand having label 0 and high bike demand with label 1.

Model	**Accuracy**	**Precision (0)**	**Precision (1)**	**Recall (0)**	**Recall (1)**
Baseline Model	0.81	0.81	0.00	**1.00**	0.00
LDA	0.85	0.89	0.56	0.95	0.37
QDA	0.88	0.92	0.71	0.94	0.66
Nearest Neighbours	0.86	**0.93**	0.64	0.90	**0.71**
Random Forest	0.88	0.91	0.72	0.95	0.58
AdaBoost	0.85	0.92	0.59	0.91	0.62
XGBoost	0.88	0.91	0.70	0.94	0.60
Deep Neural Network	**0.90**	**0.93**	**0.74**	0.94	**0.71**

9 Conclusions

We evaluated five classification models for predicting the demand for bikes. The best overall methodology was to construct an ensemble of 50 over-parameterized DNNs, which shows the best performance on most metrics, with no significant weaknesses.

References

[1] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 2011;12:2825-30. Available from: https://scikit-learn.org/stable/modules/lda_qda.html.

[2] Cover T, Hart P. Nearest neighbor pattern classification. IEEE Transactions on Information Theory. 1967;13(1):21-7.

[3] Géron A. Hands-on machine learning with Scikit-learn, Keras, and TensorFlow : concepts, tools, and techniques to build intelligent systems. 2nd ed. Sebastopol: O'Reilly; 2019. Publication Title: Hands-on machine learning with Scikit-learn, Keras, and TensorFlow : concepts, tools, and techniques to build intelligent systems.

[4] Freund Y, Schapire RE. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. Journal of Computer and System Sciences. 1997;55(1):119-39. Available from: https://www.sciencedirect.com/science/article/pii/S002200009791504X.

[5] Chen T, Guestrin C. XGBoost: A Scalable Tree Boosting System; 2016.

[6] He K, Zhang X, Ren S, Sun J. Deep Residual Learning for Image Recognition. arXiv; 2015. ArXiv:1512.03385 [cs]. Available from: http://arxiv.org/abs/1512.03385.

[7] Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv; 2015. ArXiv:1502.03167 [cs]. Available from: http://arxiv.org/abs/1502.03167.

[8] Nakkiran P, Kaplun G, Bansal Y, Yang T, Barak B, Sutskever I. Deep Double Descent: Where Bigger Models and More Data Hurt. arXiv; 2019. ArXiv:1912.02292 [cs, stat]. Available from: http://arxiv.org/abs/1912.02292.

10 Appendix

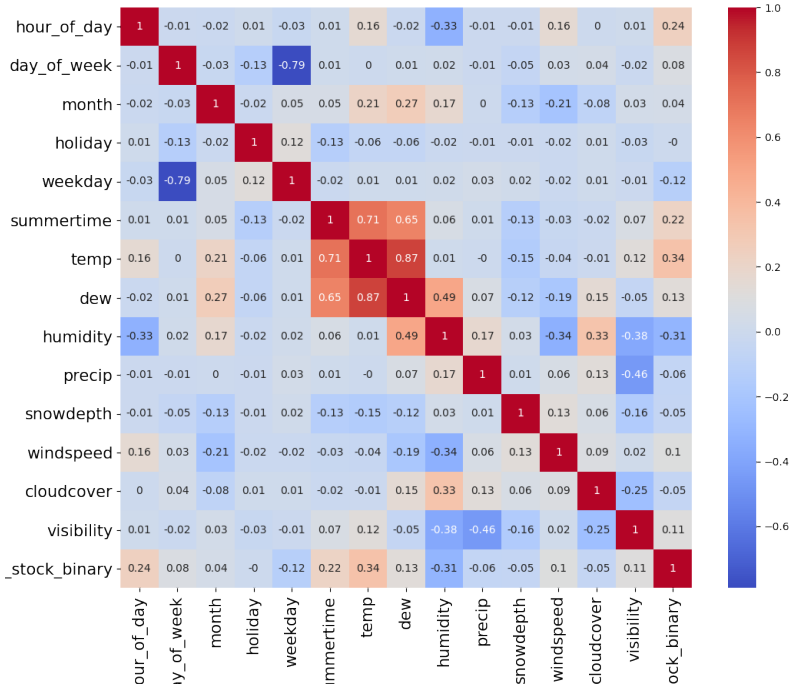

Figure 4: Correlation matrix of the features.

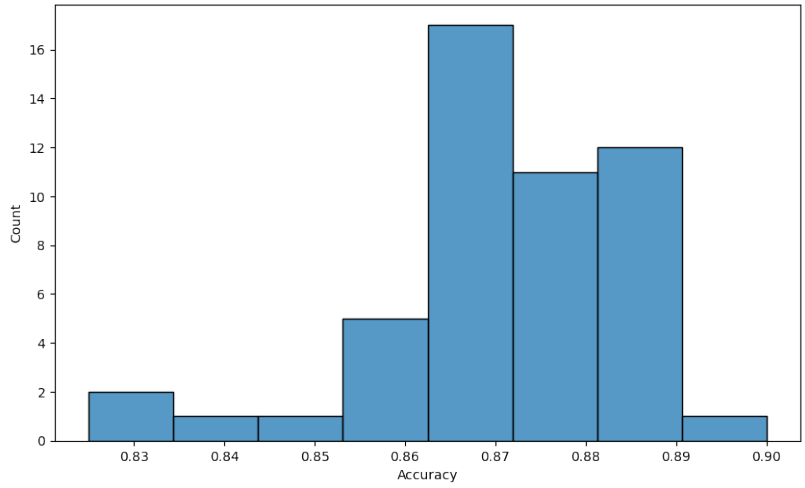

Figure 5: We show the accuracy for the 50 DNNs that form the DNN ensemble in table 2. From the accuracy distribution, we see that despite there being poor models in the ensemble, they still helped to increase the overall accuracy.

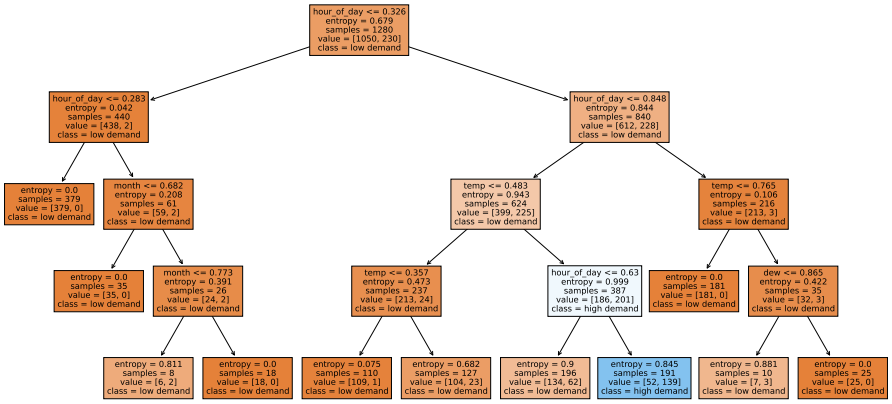

Figure 6: Visualization of a decision tree of depth 4. The RN first checks if the hour of the day (normalized) is below 0.326, if not, it creates a new DN to the right, checking for a later time in the day. If yes, it creates a DN to the left, this time it checks for the temperature of the day if it is below 0.483. This procedure is repeated until the max depth of the DT is reached. The resulting LNs contain the entropy of each prediction.

Objective Function

The XGBoost objective function consists of a loss function and a regularization term. The objective function is formulated as:

$$\text{Obj} = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{25}$$

The loss function measures the model's prediction error with respect to the true values. In XGBoost, the loss function can be the logistic loss for classification problems, squared loss for regression problems, and so on, depending on the problem at hand. For regression problems, the squared loss function is commonly used and takes the following form:

$$\ell(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2 \tag{26}$$

The regularization term is employed to control the model's complexity and prevent overfitting. The XGBoost regularization term consists of two parts: the number of leaf nodes and the sum of the squares of leaf node weights. The regularization term is expressed as:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{27}$$

Mathematical Symbol Description

Table 3: Description of Symbols in AdaBoost

Symbol	Description
D	Training dataset, where x_i is the feature vector and y_i is the label.
w_i	Weight assigned to each sample. Initialized as $\frac{1}{N}$.
T, t	Number of iterations (rounds).
h_t	Weak classifier selected at the t-th round.
ε_t	Error rate of the t-th base classifier h_t.
α_t	Weight assigned to the t-th classifier.
Z_t	Normalization factor for sample weights at the t-th iteration.
$H(x)$	Final strong classifier.
$\mathrm{I}(\cdot)$	Indicator function. Returns 1 if the condition is true, 0 otherwise.
$\mathrm{sign}(\cdot)$	Sign function. Returns -1 for negative values, 0 for zero, and 1 for positive values.

Table 4: Description of Symbols in XGBoost

Symbol	Description
Obj	Objective function
$\ell(y_i, \hat{y}_i)$	Loss function
$\Omega(f_k)$	Regularization term
n	Number of samples
K	Number of trees
T	Number of leaf nodes
w_j	Weight of a leaf node
γ	Regularization parameter
λ	Regularization parameter

Code

LDA

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
    as LDA
from sklearn.discriminant_analysis import
    QuadraticDiscriminantAnalysis as QDA
from sklearn.utils import shuffle
from sklearn.model_selection import cross_val_score

# Load the dataset
file_path = 'training_data.csv'
data = pd.read_csv(file_path)
data['increase_stock_binary'] = data['increase_stock'].apply(lambda x:
     1 if x == 'high_bike_demand' else 0)

# Manually combining weather data with only numerical values

# Creating a "Feels-Like" Temperature feature
def calculate_feels_like_temp(temp, humidity, wind_speed):
    # Constants for the formula
```

```
    c1 = -8.78469475556
    c2 = 1.61139411
    c3 = 2.33854883889
    c4 = -0.14611605
    c5 = -0.012308094
    c6 = -0.0164248277778
    c7 = 0.002211732
    c8 = 0.00072546
    c9 = -0.000003582

    apparent_temp = (c1 + (c2 * temp) + (c3 * humidity) + (c4 * temp * humidity) +
                     (c5 * temp**2) + (c6 * humidity**2) + (c7 * temp**2 * humidity) +
                     (c8 * temp * humidity**2) + (c9 * temp**2 * humidity**2))

    # Adjusting for wind speed (only if temperature is below 10 degrees Celsius)
    if temp < 10:
        wind_chill = 13.12 + 0.6215 * temp - 11.37 * wind_speed**0.16 + 0.3965 * temp * wind_speed**0.16
        return min(apparent_temp, wind_chill)
    else:
        return apparent_temp

# Apply the function to the dataset
data['feels_like_temp'] = data.apply(lambda x: calculate_feels_like_temp(x['temp'], x['humidity'], x['windspeed']), axis=1)


# Creating "workday" feature
data['workday'] = data.apply(lambda row: 1 if row['day_of_week'] < 5 and row['holiday'] == 0 else 0, axis=1)

# Daytime: day 7-18: 1, night 19 - 6: 0
data['daytime'] = data['hour_of_day'].apply(lambda x: 1 if x in np.arange(7, 19) else 0)

# Dropping original features used in composition
data = data.drop(['temp', 'humidity', 'windspeed','snow', 'dew', 'increase_stock', 'holiday', 'weekday',

                  'day_of_week','summertime', 'month', 'hour_of_day', 'cloudcover', 'month'] , axis=1)



# Displaying the modified DataFrame
data.head()


#Shuffle train data
data = shuffle(data, random_state=0)

# Preparing the dataset
X = data.drop(['increase_stock_binary'], axis=1)
y = data['increase_stock_binary']

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# split train and test data set
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
    test_size=0.2, random_state=0)

# Creating the classifier to the data
qda = QDA()

# Fitting the classifier with the training data
qda.fit(X_scaled, y)

# Prediction on test data
y_predic = qda.predict(X_test)

# Apply k-fold cross-validation
# Set the number of folds in 'cv'; for example, 5 folds
cv_scores = cross_val_score(qda, X_scaled, y, cv=10, scoring='accuracy
    ')

# Print the accuracy for each fold
print("Accuracy scores for each fold:", cv_scores)

# Print the average accuracy across all folds
print("Average accuracy E_holdout:", np.mean(cv_scores))

# Evaluation
print("Confusion Matrix:\n", confusion_matrix(y_test, y_predic))
print("\nClassification Report:\n", classification_report(y_test,
    y_predic))
print("Accuracy Score E_train:", accuracy_score(y_test, y_predic))
```

QDA

```
# Load the dataset
file_path = 'training_data.csv'
data = pd.read_csv(file_path)
data['increase_stock_binary'] = data['increase_stock'].apply(lambda x:
     1 if x == 'high_bike_demand' else 0)

# Manually combining weather data with only numerical values

# Creating a "Feels-Like" Temperature feature
def calculate_feels_like_temp(temp, humidity, wind_speed):
    # Constants for the formula
    c1 = -8.78469475556
    c2 = 1.61139411
    c3 = 2.33854883889
    c4 = -0.14611605
    c5 = -0.012308094
    c6 = -0.0164248277778
    c7 = 0.002211732
    c8 = 0.00072546
    c9 = -0.000003582

    apparent_temp = (c1 + (c2 * temp) + (c3 * humidity) + (c4 * temp *
     humidity) +
                     (c5 * temp**2) + (c6 * humidity**2) + (c7 * temp
    **2 * humidity) +
                     (c8 * temp * humidity**2) + (c9 * temp**2 *
    humidity**2))

    # Adjusting for wind speed (only if temperature is below 10
    degrees Celsius)
    if temp < 10:
        wind_chill = 13.12 + 0.6215 * temp - 11.37 * wind_speed**0.16
    + 0.3965 * temp * wind_speed**0.16
```

```
        return min(apparent_temp, wind_chill)
    else:
        return apparent_temp

# Apply the function to the dataset
data['feels_like_temp'] = data.apply(lambda x:
    calculate_feels_like_temp(x['temp'], x['humidity'], x['windspeed'
    ]), axis=1)

# Creating "workday" feature
data['workday'] = data.apply(lambda row: 1 if row['day_of_week'] < 5
    and row['holiday'] == 0 else 0, axis=1)




data = data.drop(['temp', 'humidity', 'windspeed', 'dew', 'snow', '
    increase_stock', 'cloudcover', 'visibility', 'summertime',
                  'month', 'holiday', 'weekday', 'day_of_week',  '
    precip', 'snowdepth'], axis=1)
data.head()

#Shuffle train data
data = shuffle(data, random_state=0)

# Preparing the dataset
X = data.drop(['increase_stock_binary'], axis=1)
y = data['increase_stock_binary']

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# split train and test data set
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
    test_size=0.2, random_state=0)

# Creating the classifier to the data
qda = QDA()

# Fitting the classifier with the training data
qda.fit(X_scaled, y)

# Prediction on test data
y_predic = qda.predict(X_test)

# Apply k-fold cross-validation
# Set the number of folds in 'cv'; for example, 5 folds
cv_scores = cross_val_score(qda, X_scaled, y, cv=10, scoring='accuracy
    ')

# Print the accuracy for each fold
print("Accuracy scores for each fold:", cv_scores)

# Print the average accuracy across all folds
print("Average accuracy E_holdout:", np.mean(cv_scores))

# Evaluation
print("Confusion Matrix:\n", confusion_matrix(y_test, y_predic))
print("\nClassification Report:\n", classification_report(y_test,
    y_predic))
print("Accuracy Score E_train:", accuracy_score(y_test, y_predic))
```

k-NN

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import cross_val_score
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn.model_selection import StratifiedKFold
# Load the dataset
file_path = 'training_data.csv'
data = pd.read_csv(file_path)
data['increase_stock_binary'] = data['increase_stock'].apply(lambda x:
     1 if x == 'high_bike_demand' else 0)
# Separating features and the target variable
# split daytime into 3 values 0, 0.5, 1 depending on the hour of the
    day
data['daytime'] = data['hour_of_day'].apply(lambda x: 1 if 15 <= x <=
    19 else 0.5 if 8 <= x <= 14 else 0)
#identify this day is workday or not?
def is_workday(row):
    if row['day_of_week'] >= 5:  # If the day is Saturday (5) or
    Sunday (6), it's not a workday
        return 0
    if row['holiday'] == 1:  # If it's a holiday, it's not a workday
        return 0
    return 1  # Otherwise, it's a workday

# Apply the function to the dataset
data['is_workday'] = data.apply(is_workday, axis=1)
#http://www.qxkp.net/qxbk/qxsy/202103/t20210301_2787280.html
def categorize_precipitation(mm):
    if mm < 10:
        return 1
    elif 10 <= mm < 25:
        return 2
    elif 25 <= mm < 50:
        return 3
    elif 50 <= mm < 100:
        return 4
    elif 100 <= mm < 250:
        return 5
    elif mm >= 250:
        return 6
    else:
        return 0

# Snowfall: Categorizing based on the description
def categorize_snow(snowfall, snowdepth):
    if snowfall == 0 and snowdepth == 0:
        return 0
    elif snowfall == 0 and snowdepth > 0:
        return 1
    elif snowfall > 0 and snowdepth == 0:
        return 2
    elif snowfall > 0 and snowdepth > 0:
        return 3
    else:
        return 4

# Apply the function to the rows of the dataframe
```

```
data['snow_condition'] = data.apply(lambda row: categorize_snow(row['snow'], row['snowdepth']), axis=1)

# Now let's drop the original 'snow' and 'snowdepth' columns

# Atmospheric pressure: No categories needed, it's a continuous variable

# Wind speed: The description is related to the Beaufort scale which is different from the wind speed.
# Hence, no categorization is needed. However, if there is a specific request to categorize wind speed based on
# a given scale, that could be done.

# Applying the categorization functions to the precipitation and snowfall columns
data['precip_category'] = data['precip'].apply(categorize_precipitation)
# # Apply the function to the 'windspeed' column
# data['wind_level'] = data['windspeed'].apply(categorize_wind_speed)
apparent_temp = 1.07 * data['temp'] + 0.2 * data['dew'] - 0.65 * data['windspeed'] - 2.7

# Add the calculated apparent temperatures to the dataframe
data['apparent_temp'] = apparent_temp
#season spring, summer, fall, winner
# Function to categorize months into seasons
def categorize_season(month):
    if month in [12, 1, 2]:
        return 0
    elif month in [3, 4, 5]:
        return 1
    elif month in [6, 7, 8]:
        return 2
    else:  # months 9, 10, 11
        return 3

# Apply the function to categorize seasons
data['season'] = data['month'].apply(categorize_season)

# Merge 'season' and 'summertime' into a single column
# If summertime is 1, then it overrides the season to 'Summer'
data['season_summertime'] = data.apply(lambda row: 2 if row['summertime'] == 1 else row['season'], axis=1)
#drop some features.
data = data.drop(columns=['hour_of_day', 'day_of_week', 'holiday', 'weekday','snow','month','summertime','season','temp','dew']) # windspeed fearture is very important.
# X = data.drop(['increase_stock', 'increase_stock_binary'], axis=1)
# y = data['increase_stock_binary']
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
 # split feature and target
X = data.drop(['increase_stock','increase_stock_binary'], axis=1)
y = data['increase_stock_binary']

 # split train and test data set
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, random_state=0)
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score

# Assuming X_train, X_test, y_train, y_test are already defined

# Creating a pipeline with a standard scaler and a KNN classifier
pipeline = make_pipeline(MinMaxScaler(), KNeighborsClassifier())

# Defining the parameter grid: testing different values for
    n_neighbors
param_grid = {'kneighborsclassifier__n_neighbors': range(1, 31)}

# Creating a GridSearchCV object
grid_search = GridSearchCV(pipeline, param_grid, cv=5)

# Fitting grid_search on the training data
grid_search.fit(X_train, y_train)

# Best parameters found by grid search
print("Best parameters:", grid_search.best_params_)

# Using the best model to make predictions
y_pred = grid_search.predict(X_test)

# Generating a classification report, confusion matrix, and accuracy
    score
classification_report_result = classification_report(y_test, y_pred)
confusion_matrix_result = confusion_matrix(y_test, y_pred)
accuracy_score_data = accuracy_score(y_test, y_pred)

# Printing the results
print("Classification Report:\n", classification_report_result)
print("Confusion Matrix:\n", confusion_matrix_result)
print("Accuracy Score:\n", accuracy_score_data)

# The cross-validation scores for the best model can also be accessed
print("Best Model Cross-Validation Scores:", grid_search.best_score_)
```

Boosting

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

# Load the dataset
file_path = r"Algorithms/training_data.csv"
data = pd.read_csv(file_path)
data1 = data.copy()
headers = list(data.columns)
# Display the first few rows of the dataframe
data.head()
import sklearn.preprocessing as preprocessing
from sklearn.preprocessing import MinMaxScaler

minmax = MinMaxScaler()
data_ = data.iloc[:, :-1]
minmax.fit(data_)
data = pd.DataFrame(minmax.transform(data_), columns=headers[:-1])
```

```
data["increase_stock"] = data1["increase_stock"]
data["increase_stock_binary"] = data["increase_stock"].apply(
    lambda x: 1 if x == "high_bike_demand" else 0
)
data.head()
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.model_selection import train_test_split

# set test set 20% of the original data, set random_state to ensure that the training set and test set obtained each time are the same, random_state can be set to any integer,
# as long as the value used for each training is the same
X_train, X_test, y_train, y_test = train_test_split(
    data.iloc[:, :-1], data.iloc[:, -1], test_size=0.25, random_state=0
)


gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42)
gb.fit(X_train, y_train)

# output feature importance score
print("Feature importance scores:", gb.feature_importances_)

# select features with the top 9 feature importance scores
selected_features = np.array(data.columns)[
    np.argsort(gb.feature_importances_)[::-1][:9]
]
print("Selected features:", selected_features)

select_columns = [
    "hour_of_day",
    "temp",
    "humidity",
    "dew",
    "day_of_week",
    "weekday",
    "windspeed",
    "month",
    "cloudcover",
    "increase_stock_binary",
]
data = data[select_columns]

ada = AdaBoostClassifier(random_state=0)
ada.fit(X_train, y_train)


from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier

#  create an AdaBoostClassifier
ada = AdaBoostClassifier(random_state=0)

# define the hyperparameter space to be tuned
param_grid = {"n_estimators": [50, 100, 200], "learning_rate": [0.01, 0.1, 0.2]}

```

```
# use GridSearchCV for grid search
grid_search = GridSearchCV(
    estimator=ada, param_grid=param_grid, scoring="accuracy", cv=3
)
grid_search.fit(X_train, y_train)

# output the best parameters and corresponding performance
print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy:", grid_search.best_score_)


from xgboost.sklearn import XGBClassifier


xgb = XGBClassifier(random_state=0)

xgb.fit(X_train, y_train)

from sklearn.model_selection import GridSearchCV
from xgboost.sklearn import XGBClassifier

# create XGBoost classifier
xgb = XGBClassifier(random_state=0)

# define the hyperparameter space to optimize
param_grid = {
    "n_estimators": [50, 100, 200],
    "max_depth": [3, 5, 7],
    "learning_rate": [0.01, 0.1, 0.2],
}

# use GridSearchCV for grid search
grid_search = GridSearchCV(
    estimator=xgb, param_grid=param_grid, scoring="accuracy", cv=4
)
grid_search.fit(X_train, y_train)

# output the best parameters and the corresponding performance
print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy:", grid_search.best_score_)


from sklearn.model_selection import StratifiedKFold

kf = StratifiedKFold(5, shuffle=True, random_state=0)
ac = []
model = ada
for train_index, test_index in kf.split(X_train, y_train):
    X_train_kf, X_test_kf = X_train.iloc[train_index], X_train.iloc[
    ↪ test_index]
    y_train_kf, y_test_kf = y_train.iloc[train_index], y_train.iloc[
    ↪ test_index]
    model.fit(X_train_kf, y_train_kf)
    ac.append(accuracy_score(y_test_kf, model.predict(X_test_kf)))
print(np.mean(ac))
print("as", accuracy_score(y_test, model.predict(X_test)))
print(ac)
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

```
from sklearn.metrics import r2_score
from math import sqrt

print("RMSE:", sqrt(mean_squared_error(y_test, model.predict(X_test))))
print("MAE:", mean_absolute_error(y_test, model.predict(X_test)))
print("R2:", r2_score(y_test, model.predict(X_test)))
from sklearn.metrics import classification_report

print(classification_report(y_test, model.predict(X_test)))


from sklearn.model_selection import StratifiedKFold

kf = StratifiedKFold(5, shuffle=True, random_state=0)
ac = []
model = xgb
for train_index, test_index in kf.split(X_train, y_train):
    X_train_kf, X_test_kf = X_train.iloc[train_index], X_train.iloc[
    ↪ test_index]
    y_train_kf, y_test_kf = y_train.iloc[train_index], y_train.iloc[
    ↪ test_index]
    model.fit(X_train_kf, y_train_kf)
    ac.append(accuracy_score(y_test_kf, model.predict(X_test_kf)))
print(np.mean(ac))
print("as", accuracy_score(y_test, model.predict(X_test)))
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from math import sqrt

print("RMSE:", sqrt(mean_squared_error(y_test, model.predict(X_test))))
print("MAE:", mean_absolute_error(y_test, model.predict(X_test)))
print("R2:", r2_score(y_test, model.predict(X_test)))

from sklearn.metrics import classification_report

print(classification_report(y_test, model.predict(X_test)))
# stratifiedfrom sklearn.model_selection import StratifiedKFold
kf = StratifiedKFold(5, shuffle=True, random_state=0)
ac = []
model = xgb
for train_index, test_index in kf.split(X_train, y_train):
    X_train_kf, X_test_kf = X_train.iloc[train_index], X_train.iloc[
    ↪ test_index]
    y_train_kf, y_test_kf = y_train.iloc[train_index], y_train.iloc[
    ↪ test_index]
    model.fit(X_train_kf, y_train_kf)
    ac.append(accuracy_score(y_test_kf, model.predict(X_test_kf)))
print(np.mean(ac))
print("as", accuracy_score(y_test, model.predict(X_test)))
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from math import sqrt

print("RMSE:", sqrt(mean_squared_error(y_test, model.predict(X_test))))
print("MAE:", mean_absolute_error(y_test, model.predict(X_test)))
print("R2:", r2_score(y_test, model.predict(X_test)))
# recall precise
```

```
from sklearn.metrics import classification_report

print(classification_report(y_test, model.predict(X_test)))
# stratified


from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# cross_val_results = cross_val_score(model, X_train, y_train, cv=kf,
    ↪ scoring='accuracy', fit_params={'sample_weight': None})
# use cross validation to evaluate the trained model
cross_val_results = cross_val_score(
    model, X_test, y_test, cv=kf, scoring="accuracy", fit_params={"
    ↪ sample_weight": None}
)

# output performance indicators
print("Cross-Validation Results:", cross_val_results)
print("Mean Accuracy:", cross_val_results.mean())
```

DNN

```
# !pip install optuna -q
import pandas as pd
import numpy as np
import optuna
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import StratifiedKFold
from tqdm import tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F

import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = (10, 7)
plt.rcParams["figure.dpi"] = 100

file_path = 'training_data.csv'
data = pd.read_csv(file_path)
data['increase_stock_binary'] = data['increase_stock'].map(lambda x: 0 if x
    ↪  == 'low_bike_demand' else 1)
data = data.drop(['snow', 'increase_stock'], axis=1)

# pick out the labels
y = data['increase_stock_binary'].to_numpy().astype(np.float32)
y = torch.tensor(y).unsqueeze(-1)
data = data.drop(['increase_stock_binary'], axis=1)

# Separating the numerical and categorical data to handle them separately
onehot = True
if onehot:
```

```
    cat_data = pd.DataFrame({key:data[key] for key in data.keys() if data[key].dtype == int})
    num_data = pd.DataFrame({key:data[key] for key in data.keys() if data[key].dtype == float})

    # performing onehot encoding on the categorical data
    cat_data = torch.from_numpy(cat_data.to_numpy())
    cat_onehot = torch.cat([F.one_hot(x, num_classes=24) for x in cat_data]).view(cat_data.shape[0], -1)

    # constructing the complete input dataset
    data = np.concatenate([num_data.to_numpy(), cat_onehot.numpy()], axis=-1).astype(np.float32)

# scaling
X = MinMaxScaler().fit_transform(data)
X = torch.Tensor(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, shuffle=True, random_state=0)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)

class DNN(nn.Module):
    def __init__(self, hidden_size, input_size, layers=2, seed=0, submodule=False):
        super().__init__()
        torch.manual_seed(seed)

        self.hidden_size = hidden_size
        self.input_size = input_size
        self.submodule = submodule

        self.activation = nn.ReLU()
        self.input_layer = nn.Linear(input_size, hidden_size)
        self.batch_norm_input = nn.BatchNorm1d(hidden_size)  # BatchNorm for input layer
        self.linear_layers = nn.ModuleList()
        self.batch_norm_layers = nn.ModuleList()  # BatchNorm for other layers

        for _ in range(layers):
            self.linear_layers.append(nn.Linear(hidden_size, hidden_size))
            self.batch_norm_layers.append(nn.BatchNorm1d(hidden_size))

        self.output_layer = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = self.input_layer(x)
        x = self.batch_norm_input(x)
        x = self.activation(x)

        for layer, batch_norm in zip(self.linear_layers, self.batch_norm_layers):
            residual = x
            x = layer(x)
            x = batch_norm(x)
            x = self.activation(x + residual)

```

```
        if not self.submodule:
            x = self.output_layer(x)
            x = torch.sigmoid(x)
        return x

epochs = 100
metric = lambda prediction, label: (prediction.round() == label).numpy().
    ↪ mean()
loss_fn = torch.nn.BCELoss()

model = DNN(17, X_train.shape[-1], 2)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

train_loss_history = []
test_loss_history = []
test_metric_history = []

for epoch in tqdm(range(epochs)):
    optimizer.zero_grad()
    model.train()
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss.backward()
    optimizer.step()

    model.eval()
    train_loss_history.append(loss.item())
    with torch.no_grad():
        y_pred_test = model(X_test)
        test_loss = loss_fn(y_pred_test, y_test)
        test_loss_history.append(test_loss.item())
        accuracy = metric(y_pred_test, y_test)
        if len(test_metric_history) > 0 and accuracy > max(
    ↪ test_metric_history):
            optimal_epoch = epoch
        test_metric_history.append(accuracy)


plt.figure(figsize=(10, 5), dpi=100)
plt.plot(test_metric_history, label="test accuracy")
plt.axvline(
    optimal_epoch,
    color="red",
    label=f"best accuracy {test_metric_history[optimal_epoch]:.3f} occured
    ↪ at epoch {optimal_epoch}")
plt.ylim(0.8, 0.95)
plt.legend(), plt.grid()
plt.show()


epochs = 100
loss_fn = nn.BCELoss()
n_ensembles = 10 # checking the performance for different weight
    ↪ initialization
kf = StratifiedKFold(n_splits=5, random_state=0, shuffle=True)
optimal_models = []

for n in range(n_ensembles):
    fold_performance = []
```

```
    for cv, (train_index, test_index) in enumerate(kf.split(X_train,
    ↪ y_train)):
        model = DNN(200, X.shape[-1], 2, seed=n)
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3,
    ↪ weight_decay=1)

        train_loss_history = []
        test_loss_history = []
        test_metric_history = []
        optimal_weights = 0

        for epoch in tqdm(range(epochs)):
            optimizer.zero_grad()
            model.train()
            y_pred = model(X_train[train_index])
            loss = loss_fn(y_pred, y_train[train_index])
            loss.backward()
            optimizer.step()

            with torch.no_grad():
                model.eval()
                y_pred_test = model(X_train[test_index])
                test_loss = loss_fn(y_pred_test, y_train[test_index])
                test_loss_history.append(test_loss.item())
                accuracy = metric(y_pred_test, y_train[test_index])

                if len(test_metric_history) > 0 and accuracy > max(
    ↪ test_metric_history):
                    optimal_epoch = epoch
                    optimal_weights = model.state_dict()
                test_metric_history.append(accuracy)

            train_loss_history.append(loss.item())
        optimal_models.append(optimal_weights)
        fold_performance.append(test_metric_history[optimal_epoch])

    mean_performance = np.mean(fold_performance)
    print(f"{optimal_epoch=}")
    print(f"Mean performance for ensemble {n}: {mean_performance:.3f} $\pm$
    ↪  {np.std(fold_performance).round(3)}")
    print(f"{np.round(fold_performance, 3)}")

def compute_precision(predictions, labels, positive_label=1):
    TP = ((predictions == positive_label) & (labels == positive_label)).sum
    ↪ ().item()
    FP = ((predictions == positive_label) & (labels != positive_label)).sum
    ↪ ().item()
    return TP / (TP + FP) if (TP + FP) > 0 else 0

def compute_recall(predictions, labels, positive_label=1):
    TP = ((predictions == positive_label) & (labels == positive_label)).sum
    ↪ ().item()
    FN = ((predictions != positive_label) & (labels == positive_label)).sum
    ↪ ().item()
    return TP / (TP + FN) if (TP + FN) > 0 else 0


def DeepNeuralNetworkEnsemble(optimal_models, test_data):
    DNN_ensemble = []
```

```
    for weights in optimal_models:
        model = DNN(200, X.shape[-1], 2)
        model.load_state_dict(weights)
        y_pred = model(test_data)
        DNN_ensemble.append(y_pred.detach().numpy())
    return torch.Tensor(DNN_ensemble).mean(0).round()

prediction = DeepNeuralNetworkEnsemble(optimal_models, X_test)
accuracy = metric(prediction, y_test)

print(round(accuracy, 2))
for positive_label in [0, 1]:
    recall = compute_recall(prediction, y_test, positive_label)
    precision = compute_precision(prediction, y_test, positive_label)
    print(round(precision, 2), round(recall, 2))

single_DNN_performance = []
for weights in optimal_models:
    model = DNN(200, X.shape[-1], 2)
    model.load_state_dict(weights)
    y_pred = model(X_test)
    single_DNN_performance.append(metric(y_pred, y_test))

plt.figure(figsize=(10, 6), dpi=100)
sns.histplot(single_DNN_performance)
plt.xlabel("Accuracy")
#plt.savefig("Accuracy distribution for DNN ensemble", bbox_inches="tight")
```

Random Forest

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import KFold, StratifiedKFold
from tqdm import tqdm

import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = (10, 7)
plt.rcParams["figure.dpi"] = 100

# Load the dataset
file_path = 'training_data.csv'
data = pd.read_csv(file_path)
data['increase_stock_binary'] = data['increase_stock'].apply(lambda x: 1 if
    ↪  x == 'high_bike_demand' else 0)
data = data.drop(["increase_stock"], axis=1)

X = data.to_numpy()
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

y = X[:,-1].astype(np.float32)
X = X[:,:-1].astype(np.float32)

```

```
metric = lambda a, b: (a == b).mean()

def RandomForestEnsemble(x, models):
    predictions = [model.predict_proba(x) for model in models]
    proba1 = np.mean(predictions, axis=0)[:, 1] # probability of predicting label 1
    return proba1.round() # [0,0,0,1,0,1,0,0,0,...]

def compute_precision(predictions, labels, positive_label=1):
    TP = ((predictions == positive_label) & (labels == positive_label)).sum().item()
    FP = ((predictions == positive_label) & (labels != positive_label)).sum().item()
    return TP / (TP + FP) if (TP + FP) > 0 else 0

def compute_recall(predictions, labels, positive_label=1):
    TP = ((predictions == positive_label) & (labels == positive_label)).sum().item()
    FN = ((predictions != positive_label) & (labels == positive_label)).sum().item()
    return TP / (TP + FN) if (TP + FN) > 0 else 0

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, random_state=0, shuffle=True)

seeds = range(50)
scores = []
models = []
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)

for s in seeds:
    # Note how the indices between the validation set and train set are flipped.
    # This means that the classifier will be trained on less than 50% of the data.
    for i, (train_index, test_index) in enumerate(kf.split(X_train, y_train )):
        clf = RandomForestClassifier(
            n_estimators=200, max_depth=100, criterion="entropy",
            random_state=s, n_jobs=-1
        )
        clf.fit(X_train[train_index], y_train[train_index])
        # accuracy = clf.score(X_train[test_index], y_train[test_index])
        accuracy = clf.score(X_test, y_test)
        scores.append(accuracy)
        models.append(clf)

print("average accuracy", np.mean(scores))

X_pred = RandomForestEnsemble(X_test, models)
metric(X_pred, y_test)

X_pred = np.zeros(len(y_test))
print(metric(X_pred, y_test))
for positive_label in [0, 1]:
    recall = compute_recall(X_pred, y_test, positive_label)
    precision = compute_precision(X_pred, y_test, positive_label)
    print(round(precision, 2), round(recall, 2))

```

```
X_pred = RandomForestEnsemble(X_test, models)
print(metric(X_pred, y_test))
for positive_label in [0, 1]:
    recall = compute_recall(X_pred, y_test, positive_label)
    precision = compute_precision(X_pred, y_test, positive_label)
    print(round(precision, 2), round(recall, 2))

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75,
    ↪ random_state=0, shuffle=True)

clf = RandomForestClassifier(
    n_estimators=200, max_depth=100, criterion="entropy", oob_score=True,
    random_state=0, n_jobs=-1
)

clf.fit(X_train, y_train)

seeds = range(10)
# kf = StraKFold(5, random_state=0, shuffle=True)

for i, (train_index, test_index) in tqdm(enumerate(kf.split(X_train,
    ↪ y_train))):
    X_, X_test = X[train_index], X[test_index]
    y_, y_test = y[train_index], y[test_index]
    models = []

    # This loop creates a list of 3 * seeds classifiers using Bagging,
    # by splitting the training set into 3 sets.
    for s in seeds:
        # Note how the indices between the validation set and train set are
    ↪  flipped.
        # This means that the classifier will be trained on less than 50%
    ↪ of the data.
        for i, (train_index_, _) in enumerate(KFold(n_splits=3).split(X_)):
            X_train = X_[train_index_]
            y_train = y_[train_index_]
            clf = RandomForestClassifier(
                n_estimators=200, max_depth=100, criterion="entropy",
                random_state=s, n_jobs=-1
            )
            clf.fit(X_train, y_train)
            models.append(clf)

    X_pred = RandomForestEnsemble(X_test, models)
    print("ensemble accuracy", metric(X_pred, y_test))

plt.hist(scores)
```