Index

Final Report

Team 7

1. Data Exploration

1-1. Descriptive Statistics

To conduct a more efficient data analysis, we examined the statistical information and general characteristics of the data using the 'describe()' function. The analysis [1] revealed that for the 'pdays' feature, the 25th, 50th, and 75th percentiles all have the value of 999, indicating that most entries are filled with 999. Additionally, the statistics for the 'age' feature indicate that a significant portion of the bank's customers are middle-aged. Also, since the ranges of the columns vary significantly, we need to normalize and standardize the data to match the scales, aiming to enhance the model's performance.

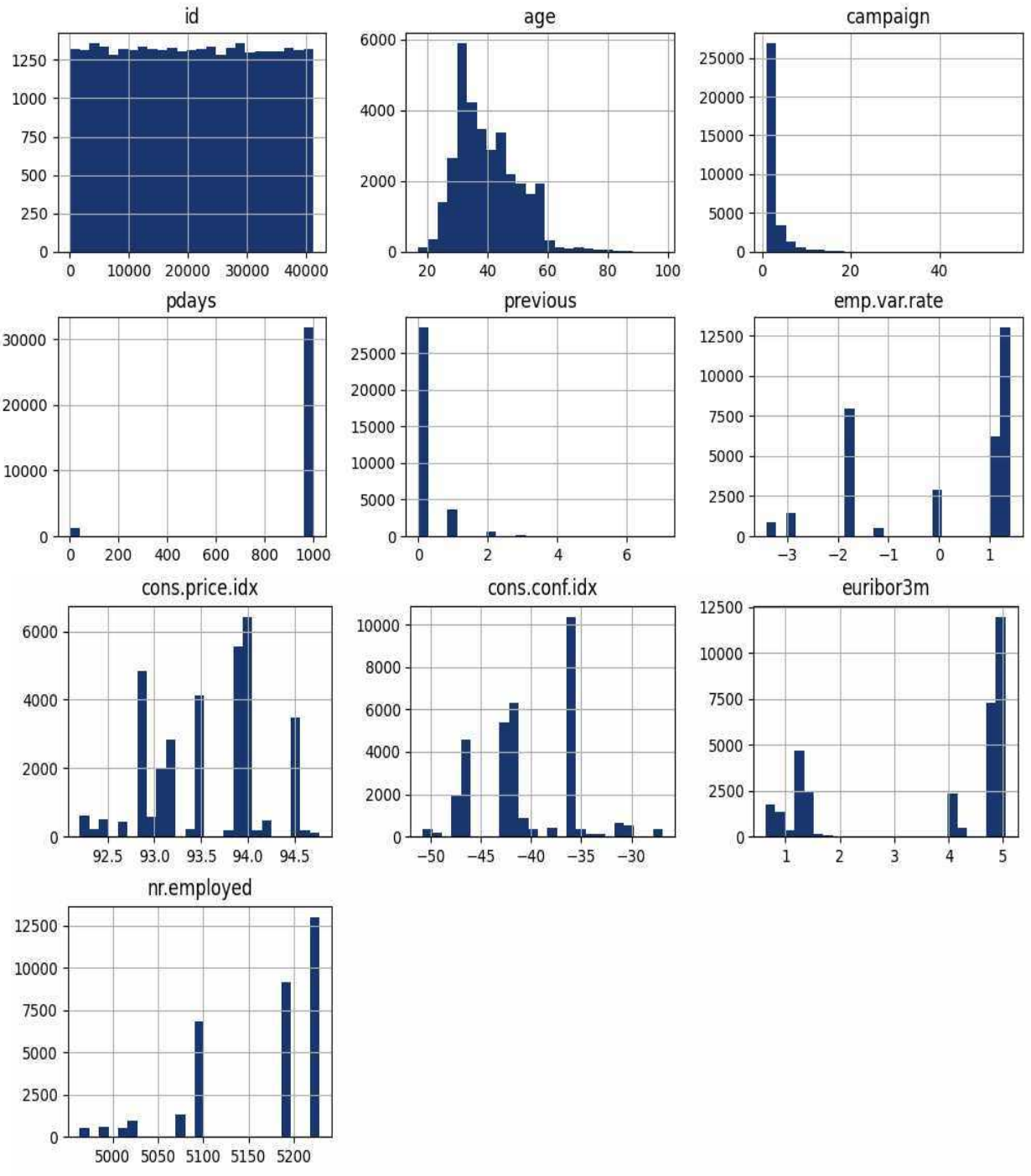
(picture 1)

	count	mean	std	min	25%	50%	75%	max
id	32950.000000	20569.615569	11895.520420	1.000000	10258.250000	20571.000000	30846.750000	41188.000000
pdays	32950.000000	962.415964	187.054556	0.000000	999.000000	999.000000	999.000000	999.000000
nr.employed	32950.000000	5167.036495	72.250873	4963.600000	5099.100000	5191.000000	5228.100000	5228.100000
age	32950.000000	40.023703	10.401749	17.000000	32.000000	38.000000	47.000000	98.000000
cons.conf.idx	32950.000000	-40.500091	4.632363	-50.800000	-42.700000	-41.800000	-36.400000	-26.900000
campaign	32950.000000	2.567830	2.766994	1.000000	1.000000	2.000000	3.000000	56.000000
euribor3m	32950.000000	3.622516	1.734791	0.634000	1.344000	4.857000	4.961000	5.045000
emp.var.rate	32950.000000	0.083129	1.571951	-3.400000	-1.800000	1.100000	1.400000	1.400000
cons.price.idx	32950.000000	93.576610	0.578725	92.201000	93.075000	93.749000	93.994000	94.767000
previous	32950.000000	0.172838	0.498098	0.000000	0.000000	0.000000	0.000000	7.000000

1-2. Data Visualization

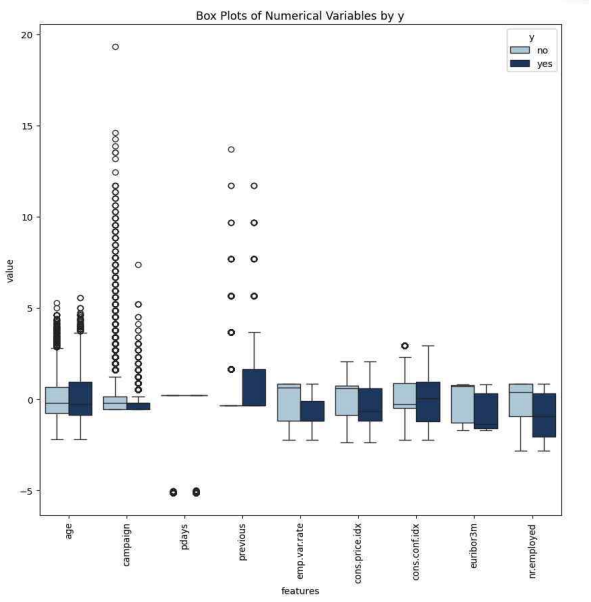
First, we plotted histograms [2] of the numerical variables to examine their distributions. As observed in the previous statistics, the 'pdays' feature has an excessively high frequency of a specific value, and the customers' ages are predominantly distributed in their 30s and 40s.

(picture 2)

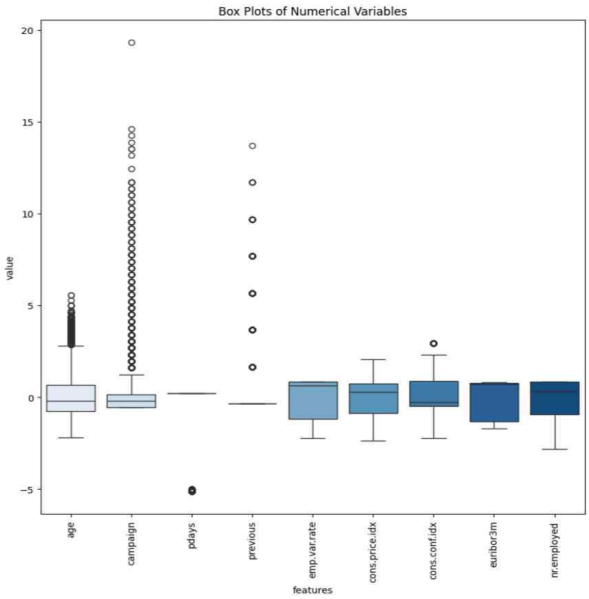


As observed in the histograms of 'campaign', 'pdays', and 'previous', there were noticeable skewed distributions in the data, leading to predictions of the presence of outliers in the dataset. To confirm this, box plots were drawn based on 'y' values[3-1] and overall[3-2]. The results revealed prominent outliers in variables with skewed distributions and outliers were also present in other variables, leading to further considerations on handling outliers.

(picture 3-1)

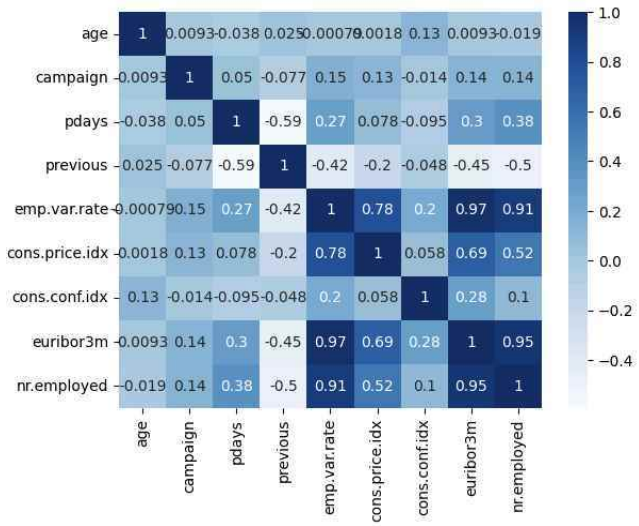


(picture 3-2)

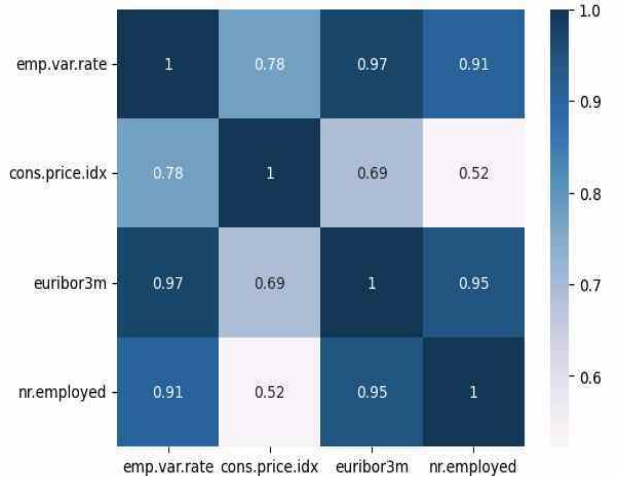


By examining the correlations between numerical variables using a heatmap[4-1], we found very high correlations among certain social and economic context attributes, so we created an additional heatmap[4-2] to verify this. Specifically, 'euribor3m' and 'emp.var.rate' had a very high correlation of 0.97. Additionally, 'nr.employed' and 'euribor3m' had a high correlation coefficient of 0.95, and 'nr.employed' and 'emp.var.rate' had a high correlation coefficient of 0.91. This indicates the need for measures to reduce multicollinearity later.

(picture 4-1)

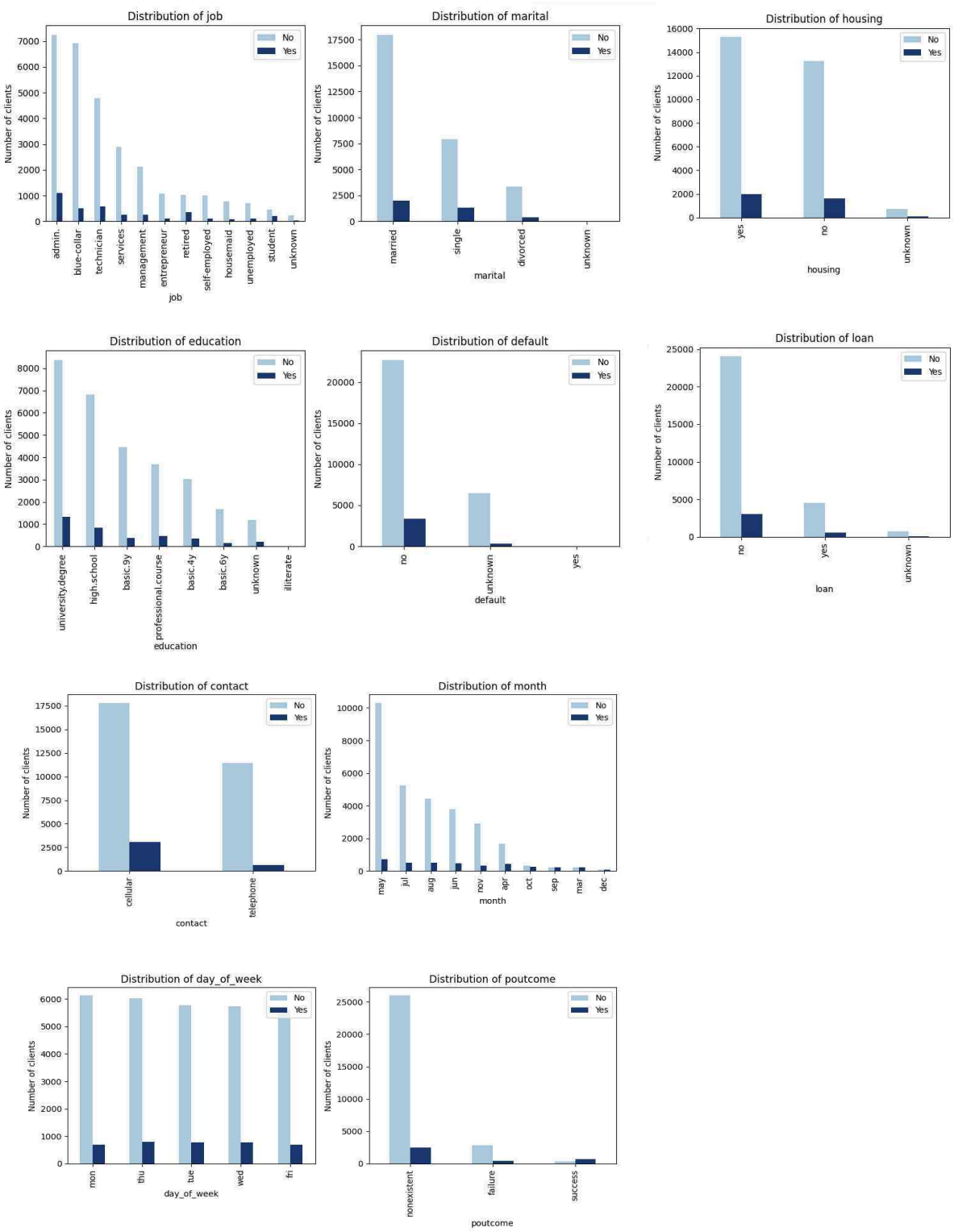


(picture 4-2)



To gain a better understanding of the categorical data, we plotted bar plots [5] based on the target value (y).

(picture 5)



2. Data pre-processing

2-1. Feature Selection

First, the 'id' column was removed as it does not hold meaningful information. The 'default' column was dropped because it has a missing value rate exceeding 20%, and one of its unique values, 'yes,' appears only three times. The 'contact' column was eliminated after observing that the model's performance improved when this feature, which had low feature importance during training, was removed. The 'day_of_week' column was dropped as the bar plot for this categorical variable showed similar distributions of 'yes' and 'no' responses across all days of the week. Upon further analysis, the subscription rates for each day were found to be similar, ranging between 10-11%. Additionally, the 'pdays' column was removed as the majority of its values are 999, indicating clients who were not contacted in a previous campaign. This refers to customers who have never been contacted in the previous campaign, but it interferes with other values by representing it in a numerical form of 999. Also, we decided to remove the 'cons.conf.idx' variable because its correlation with the target variable (y) shows only slight differences at specific indices, but overall, there is no significant difference. Finally, the 'emp.var.rate' column, which showed low feature importance and multicollinearity with 'nr.employed' and 'euribor3m', was removed.

2-2. Handling Outliers

In outlier detection, values that lie 1.5 times the interquartile range (IQR) away from the quartiles were defined as outliers, and these values were replaced with the median. One of the outlier detection techniques is the z-score method, but it requires normalization to be applied. Since we conducted normalization afterward, we did not opt for the outlier detection method using the z-score. Additionally, we tried replacing the detected outliers with NaN values and then handling them using the missing value imputation model. However, this approach did not perform as well as the method we ultimately adopted, which involved replacing outliers with the median value. The model's performance was better when outliers were replaced with the median. The reason for performing outlier treatment before missing value imputation is that we used the 'IterativeImputer' model for handling missing values, and within its parameters, we employed the 'AdaBoostRegressor' as the estimator. AdaBoost is sensitive to noise and outliers, so we addressed the outliers first to ensure the accuracy and stability of the missing value imputation process.

2-3. Encoding

We were contemplating whether to choose 'One-Hot Encoding' or 'Label Encoding' to encode the categorical variables. The categorical variables include 'job', 'marital', 'education', 'housing', 'loan', 'month', 'day_of_week', and 'poutcome'. When considering unique types after removing unknown values for each category, there are 12 categories for 'job', 3 for 'marital', 7 for 'education', 2 for 'housing', 2 for 'loan', 10 for 'month', 5 for 'day_of_week', and 2 for 'poutcome'. To find the encoding method that fits this data set, we will describe the characteristics of each encoding.

1. Label Encoding

- Memory usage is low because each category is converted to a single number
- simple to apply even when there are a lot of categories
- The order is given.
- In some models, such ordered data can be interpreted as having an order, affecting model performance.

2. One-Hot Encoding

- It is transformed into an independent vector so that no order problem arises.
- Many linear models and neural networks handle one-hot encoded data well.
- If there are many categories, a very large sparse matrix can be generated (the curse of dimensions)
- High-dimensional data can make the training of the model more difficult and slow.

Considering the characteristics of each encoding method, Label encoding is suitable when there is a natural order between categories. One-Hot encoding is suitable when there is no order between categories. In addition, tree-based models can handle label encoding well, whereas linear models and neural networks perform better with One-Hot Encoded data. Thus, we're not significantly impacted since variables like education, month, and day_of_week exhibit ordered categories, and 'housing', 'loan', and 'poutcome' are treated as binary labels 0 or 1 in Label Encoding.

However, for unordered variables like 'occupation' and 'marriage', it's essential to determine whether Label or One-Hot Encoding is more suitable. Before running the model, we conducted two trials: In Trial 1 [6] only 'occupation' and 'marriage' variables were One-Hot encoded, while in

Trial 2, the entire dataset was Label encoded. Model performance was better during Trial 2 than Trial 1. This is thought to have been affected by increased model complexity when the encoding was not unified.

Therefore, the encoding process was carried out by unifying it with label encoding.

(picture 6 : Trial 1 - combine two encoding method)

```
0초
1 df.columns

Index(['age', 'campaign', 'previous', 'cons.price.idx', 'euribor3m',
       'nr.employed', 'education', 'housing', 'loan', 'month', 'poutcome',
       'job_admin.', 'job_blue-collar', 'job_entrepreneur', 'job_housemaid',
       'job_management', 'job_retired', 'job_self-employed', 'job_services',
       'job_student', 'job_technician', 'job_unemployed', 'marital_divorced',
       'marital_married', 'marital_single', 'y'],
      dtype='object')
```

2-4. Handling Missing Values

Initially, upon inspecting the dataset, there appeared to be no missing values. However, it was later discovered that missing values were represented by 'unknown' and 'nonexistent' instead of Null values, making them difficult to identify. Therefore, these values were converted to Null values to accurately identify the missing data. Upon re-evaluation, it was found that a significant amount of missing data existed. At first, we attempted to use a missing value prediction model by one-hot encoding the categorical variables with missing values. However, after encoding, using KNN or iterative models did not yield successful results in predicting the missing values. Moreover, label encoding treated missing values as unique values, assigning them labels and causing the missing values to disappear, which posed a problem.

Ultimately, after several attempts, we decided to apply label encoding to the categorical variables (excluding those with missing values) and then use a prediction model. Initially, we used the KNN imputer, but it was computationally intensive, resulting in long processing times and memory inefficiencies, such as session crashes. Therefore, we switched to the iterative imputer, which replaces missing values using a regression model. After experimenting with different estimators, we found that 'AdaBoostRegressor' performed the best, so we finalized the parameters using this estimator.

2-5. PCA (Principal Component Analysis)

From the previous heatmap figure, you can see that nr. deployed, euribor3m is very correlated at 0.95. Since there are the highest emp.var.rate and Euribor3m, but emp.var.rate is not selected in the variable selection, we will use PCA to reduce the multicollinearity of nr. deployed and euribor3m, which is the second highest, and replace them with one new variable by reducing the dimensionality of data. PCA(Principal Component Analysis) is a statistical technique that reduces the dimensionality of data. It is used to emphasize the important characteristics of data and reduce noise. PCA finds a new axis called the principal component of the data and projects the data onto it. This axis represents the direction in which the data variance is greatest. Then, we transform the data into a new dimension by selecting the most important principal components. In this way, we can solve the multicollinearity problem by simplifying the data and at the same time making two variables with a high correlation coefficient into one variable.

2-6. Data Normalization and Scaling

Upon reviewing the statistics, it became clear that the scales of the data varied significantly, necessitating normalization and standardization. Initially, we applied normalization and standardization before splitting the data, but we realized this could lead to data leakage. This situation is similar to training with prior knowledge of the test data, potentially causing overfitting. Therefore, we decided to split the data first and then apply normalization and standardization based on the training data, ensuring consistency between the training and test data.

3. Modeling Strategies

3-1. Train-Test Split for model training

After preprocessing all the data, the data set was split into training and test sets, with 'shuffle' set to True to shuffle the data before splitting. The 'test_size' was empirically determined to be 0.18, chosen from values ranging between 0.1 and 0.5. From the data set named as 'df', the independent variables excluding the dependent variable were assigned to 'X', while the dependent (target) variable was assigned to 'y'. The data was then split using the 'train_test_split' function, with the resulting sets assigned to 'X_train', 'X_test', 'y_train', and 'y_test'. The 'random_state' was set to 100.

3-2. Model Selection

(cf. Full implementation codes for all models are in the appendix)

At the visualization and data exploration stages, we discussed which model to choose for the train on the 'train.csv', which contains both numerical and categorical data. We initially decided to choose logistic regression or decision tree based on what we had learned in class.

However, as we learned and practiced various models during the course more deeply, we realized that there might be better options beyond our initial considerations. To explore this further, we tried all the models covered in lab sessions to assess each model's performance.

(Implemented model : Logistic regression, DecisionTree, Bagging, Randomforest, AdaBoost, Gradient Boosting, XG Boost, LightGBM, Perceptron, MLP, Polynomial regression, Gaussian, SVM)

1. ***Consider characteristics of each model - according to the base model***

- Linear regression-based models

  They are simple and easy to interpret, suitable for modeling linear relationships in data. However, they may struggle to capture nonlinear relationships and adapt to high-dimensional data. (ex. logistic regression, polynomial regression, Perceptron…)

- Tree-Based Models

  Tree-Based models partition data to create a tree structure and learn decision rules at each node. These models offer interpretability, effectively handle nonlinear relationships, and facilitate feature selection without regularization or feature scaling. (ex. Decision Tree, Bagging, Random Forests, AdaBoost, GB, XGB, Light GBM)

- Neural Network Models

  Neural Network models mimic the brain's neuron structure to learn from train data through multiple layers of neurons. They consist of input layers, hidden layers, and output layers, learning complex patterns through nonlinear transformations. While capable of solving

complex problems and automatically learning features, they entail high computational costs, and are susceptible to overfitting due to numerous parameters. (ex. MLP)

- Kernel-based models

  Kernel-based models are useful for modeling nonlinear relationships and can capture complex patterns in data using various kernel functions. These models are robust to outliers and can effectively capture intricate relationships in data, but they come with high computational costs and risk of overfitting. (ex. SVM)

Given that it's a binary classification problem with a mix of numerical and categorical data, and the data exhibits nonlinearity, I anticipated that tree-based models or neural network models like MLP would be suitable.

2. ***Based on various indicators***

   1. Evaluation Indicators (Accuracy, F1, ROC-AUC, MSE)

   After training our models and performing cross-validation, we compared these metrics to select the best-performing model. First, three models with the highest values for each indicator were selected, and then the values for each indicator, including the **main metric f1 score**, were **comprehensively** considered and used for model selection.

   - ***Accuracy*** : GB(0.97), XGB(0.97), DecisionTree(0.97)
   - ***F1 score*** : 1. Perceptron(0.93) / 2. GB(0.88), XGB(0.88)
   - ***ROC-AUC*** : 1. GB(0.92) / 2. Randomforest(0.91) / 3.LGBMRegressor(0.91)
   - ***MSE***: 1. XGB(0.025) / 2. GB(0.026) / 3. Randomforest(0.027)

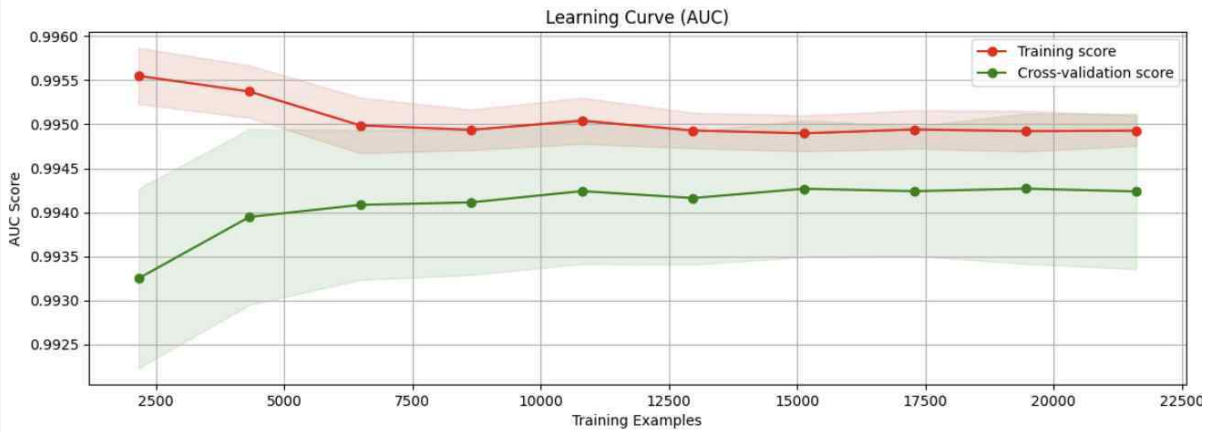
   2. Hyperparameter tuning (CV score)

   Hyperparameters are crucial elements in the model training process, directly impacting the model's performance and generalization ability. But it is really hard to know the best hyperparameter by substituting values one by one for each parameter. So, by using GridSearchCV, we were able to find the optimal hyperparameters.

So, we set the parameter grid and tried to find out the best parameters by using the function named GridSearchCV. Also, we trained the train data set through grid_search.fit function and we were able to check the best values of each parameter and the best CV score. This score is used to evaluate how well a model can generalize to new data.

On Gradient Boosting, we found out the best combination of the parameters('learning rate': 0.1, 'max_depth' : 3, 'n_estimators' : 50) and got great CV score(-0.017755)

3. *Verify problems for underfitting and overfitting*



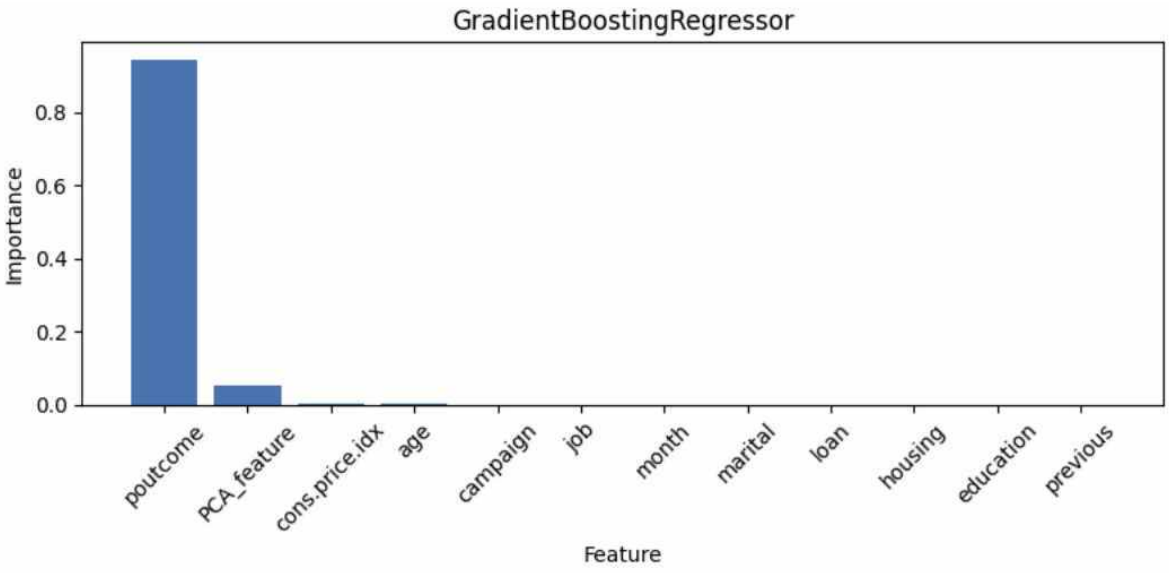
To determine whether the model has been appropriately trained, we plotted the learning curve to observe how performance changes as training progresses. The training score indicates how well the model performs on the training data, while the cross-validation score reflects the model's performance on the validation data. As shown in the graph, both the training score and the cross-validation score converge to similar values, each reaching a high value of 0.99xx. The small difference between the two scores and their convergence to similar values indicate that the model has been well-trained.

As a result of selecting a model by comprehensively judging the evaluation index, which was the most important consideration, as well as the characteristics and learning curve of each model, ***Gradient Boosting*** was finally selected.

4. Interpretation and Analysis of Results

4-1. Feature Importance

Understanding the importance of each feature is crucial for interpreting the model and identifying the factors that influence predictions. The importance of the variables is as follows:



Although variable importance slightly differed among models before model selection, the order of variable importance was generally similar across multiple models. 'poutcome' and 'PCA_feature' consistently ranked first and second in all the models that we tried. Despite the high variable importance of 'poutcome', which has a missing rate of 84% in the train data, there was concern about potential overfitting during the 'IterativeImputer' process. However, upon examining the distribution of the test data, the 'poutcome' variable showed a similar missing rate, indicating that the distributions of the train and test data were similar. This suggests that the preprocessing for the test data, our target dataset, was appropriately performed.

The next most important variable, PCA_feature, showed high variable importance for both 'nr.employed' and 'euribor3m' before applying PCA. The reduced feature importance after PCA indicates that multicollinearity has been addressed.

5. Results and Discussions

5-1. Summary of Findings

In this report, we explored various machine learning models to predict the subscription to term deposits. Key steps included data exploration, preprocessing, modeling, and result interpretation. Here are the key findings from each stage:

1. Data Exploration:
    - Descriptive Statistics: The 'pdays' feature had most values at 999, and the 'age' feature indicated a predominance of middle-aged customers. The data required normalization and standardization due to varying scales.
    - Data Visualization: Histograms and box plots revealed skewed distributions and outliers, particularly in 'campaign', 'pdays', and 'previous'. Heatmaps highlighted high correlations among economic context attributes, indicating multicollinearity.
2. Data Preprocessing:
    - Feature Selection: Irrelevant or highly sparse features such as 'id' and 'default' were removed. PCA was used to address multicollinearity and reduce feature dimensions.
    - Handling Outliers and Missing Values: Iterative Imputer was applied to handle missing values, and outliers were managed based on IQR.
    - Normalization and Scaling: Data normalization and standardization were applied post train-test split to prevent data leakage and enhance model performance.
3. Modeling Strategies:
    - Various models (Logistic Regression, Decision Tree, Bagging, Random Forest, AdaBoost, Gradient Boosting, XGBoost, LightGBM, Gaussian Process, Neural Network) were compared based on their characteristics and performance metrics.
    - Model Selection: Gradient Boosting was ultimately selected due to its high performance across accuracy, F1 score, and ROC-AUC score.
    - Hyperparameter Tuning: GridSearchCV was used for optimizing hyperparameters, leading to improved model performance.
4. Interpretation and Analysis of Results:
    - Feature Importance:'poutcome' and 'PCA_feature' consistently ranked highest in terms of the feature importance. In preprocessing steps, the imputed ‘poutcome’ feature has concerns about negative impact on the model, but we found out that the distribution pattern between test data and train data is similar.

5-2 Derived Insights

1. Model Performance:
    - Gradient Boosting provided the best balance of performance metrics, proving robust in handling both categorical and numerical data.
    - Hyperparameter tuning significantly improved model accuracy, F1 score, and ROC-AUC score, indicating the importance of this step in the modeling process.
2. Feature Importance:
    - The consistent importance of 'poutcome' and 'PCA_feature' highlights the significance of past campaign outcomes and principal component features derived from 'nr.employed' and 'euribor3m'.
3. Data Preprocessing:
    - Proper handling of missing values and outliers is critical for accurate model predictions. The use of Iterative Imputer and targeted outlier management strategies were effective in this regard.
    - Ensuring data normalization and standardization post train-test split is essential to avoid data leakage and improve model generalization.
4. Practical Applications in Financial Marketing:
    - The identified key features can guide targeted marketing campaigns, focusing on clients with certain past outcomes and specific economic contexts.

5-3. Future Directions

1. Feature Engineering:
    - Investigate additional feature engineering techniques to derive new predictive features, particularly from economic and client behavior data.
2. Real-time Implementation:
    - Develop a real-time prediction system integrating the trained model with the bank's marketing platform to provide instant insights and recommendations.
    - Implement continuous learning mechanisms to update the model with new data, ensuring it remains relevant and accurate over time.

By leveraging these insights and strategies, financial institutions can enhance their marketing efforts, leading to increased client engagement and higher subscription rates for term deposits.

6. Appendix - modeling code

6.1. Regressor

6.1.1 DecisionTree

```
[26]
# DecisionTree Regressor
model = DecisionTreeRegressor(random_state=random_state)

# Define the hyperparameters and their possible values
param_grid = {
    "max_depth": [5, 10, 20],
    "min_samples_split": [2, 10, 20],
    "ccp_alpha": [0.0, 0.01],
}

grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(grid_search.best_score_))

models["Decision Tree Regressor"] = grid_search.best_estimator_

y_pred = grid_search.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

print('')
print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.2. Bagging

```
17
# Bagging Regressor
base_model = DecisionTreeRegressor()
model = BaggingRegressor(estimator=base_model,
                         bootstrap=True,
                         n_jobs=-1,
                         random_state=random_state)

# Define the hyperparameters and their possible values
param_grid = {
    "n_estimators": [25, 50]
}

grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(grid_search.best_score_))

models["Bagging Regressor"] = grid_search.best_estimator_

y_pred = grid_search.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

print("")
print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print("")
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.3. RandomForest

```
[31]  1 # Randomforest regressor
43초  2 model = RandomForestRegressor(max_depth=None,
      3                               min_samples_split=2,
      4                               bootstrap=True,
      5                               n_jobs=-1,
      6                               random_state=random_state)
      7
      8 # Define the hyperparameters and their possible values
      9 param_grid = {
     10     "n_estimators": [25, 50],
     11     "max_features": [0.5, "sqrt", "log2", None],
     12 }
     13
     14 grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
     15 grid_search.fit(X_train, y_train)
     16
     17 print("Best parameters: ", grid_search.best_params_)
     18 print("Best CV score: {:.6f}".format(grid_search.best_score_))
     19
     20 models["Random Forest Regressor"] = grid_search.best_estimator_
     21
     22 y_pred = grid_search.predict(X_test)
     23 y_pred = (y_pred > 0.5).astype(int)
     24
     25 print('')
     26 print("Accuracy:", accuracy_score(y_test, y_pred))
     27 print("F1:", f1_score(y_test, y_pred))
     28 print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
     29 print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.4. Adaboost

```
 1 # AdaBoostRegressor
 2 model = AdaBoostRegressor(loss="linear",
 3                           random_state=random_state)
 4
 5 # Define the hyperparameters and their possible values
 6 param_grid = {
 7     "n_estimators": [25, 50],
 8     "estimator": [DecisionTreeRegressor(max_depth=3), DecisionTreeRegressor(max_depth=6)],
 9     "learning_rate": [0.1, 1.0],
10 }
11
12 grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
13 grid_search.fit(X_train, y_train)
14
15 print("Best parameters: ", grid_search.best_params_)
16 print("Best CV score: {:.6f}".format(grid_search.best_score_))
17
18 models["AdaBoost regressor"] = grid_search.best_estimator_
19
20 y_pred_adaboost = grid_search.predict(X_test)
21 y_pred_adaboost = (y_pred > 0.5).astype(int)
22
23 print('')
24 print("Accuracy:", accuracy_score(y_test, y_pred_adaboost))
25 print("F1:", f1_score(y_test, y_pred_adaboost))
26 print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
27 print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.5. Gradient Boosting

```
[33]
# GB - regressor
model = GradientBoostingRegressor(loss="squared_error",
                                  subsample=1.0,
                                  random_state=random_state)

# Define the hyperparameters and their possible values
param_grid = {
    "n_estimators": [25, 50],
    "max_depth": [3, 6],
    "learning_rate": [0,0, 0.1]
}

grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(grid_search.best_score_))

models["Gradient Boosting regressor"] = grid_search.best_estimator_
model_selected = grid_search.best_estimator_

y_pred = grid_search.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

print('')
print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.6. XG Boosting

```
# XGB
model = XGBRegressor(subsample=1.0,
                     learning_rate=0.1,
                     max_depth=7,
                     n_jobs=-1,
                     random_state=random_state)

# Define the hyperparameters and their possible values
param_grid = {
    "n_estimators": [25, 50],
    "reg_alpha": [0, 0.1],
    "reg_lambda": [0, 0.1],
}

grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True)
grid_search.fit(X_train, y_train)

print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(grid_search.best_score_))

models["XGBoost_regressor"] = grid_search.best_estimator_

y_pred = grid_search.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

print('')
print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.1.7. LightGBM

```
[43]  1 # LGBMRegressor
      2
      3 model = LGBMRegressor(learning_rate=0.1,
      4                       data_sample_strategy="goss",
      5                       top_rate=0.2,
      6                       other_rate=0.1,
      7                       force_col_wise=True,
      8                       verbosity=0,
      9                       n_jobs=-1,
     10                       random_state=random_state)
     11
     12
     13 # Define the hyperparameters and their possible values
     14 param_grid = {
     15     "n_estimators": [25, 50],
     16     "reg_alpha": [0, 0.1],
     17     "reg_lambda": [0, 0.1],
     18     "enable_bundle": [True, False]
     19 }
     20
     21 grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True)
     22 grid_search.fit(X_train, y_train)
     23
     24 print("Best parameters: ", grid_search.best_params_)
     25 print("Best CV score: {:.6f}".format(grid_search.best_score_))
     26
     27 models["LightGBM_Regressor"] = grid_search.best_estimator_
     28
     29 y_pred = grid_search.predict(X_test)
     30 y_pred = (y_pred > 0.5).astype(int)
     31
     32 print('')
     33 print("Accuracy:", accuracy_score(y_test, y_pred))
     34 print("F1:", f1_score(y_test, y_pred))
     35 print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
     36 print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.2. Gaussian

```
[53]  1 # Gaussian
      2 model = GaussianNB()
      3
      4 param_grid = {
      5     'var_smoothing': [1e-9, 1e-8, 1e-7]
      6 }
      7
      8 grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
      9 grid_search.fit(X_train, y_train)
     10
     11 print("Best parameters: ", grid_search.best_params_)
     12 print("Best CV score: {:.6f}".format(grid_search.best_score_))
     13
     14
     15 models["Gaussian Naive Bayes"] = grid_search.best_estimator_
     16
     17 y_pred = grid_search.best_estimator_.predict(X_test)
     18
     19 # Evaluate the model
     20 print("")
     21 print("Accuracy:", accuracy_score(y_test, y_pred))
     22 print("F1:", f1_score(y_test, y_pred))
     23 print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.3. Neural Network - set warmup, dropout, early stopping

```
# fix randomseed
seed_value = 100
np.random.seed(seed_value)
tf.random.set_seed(seed_value)
random.seed(seed_value)

# warmup
def warmup(epoch, lr):
    if epoch < 10:  # increase learning rate linearly for the first 10 epochs
        return lr + 0.0001
    else:
        return lr

# create model
def create_model():
    model = Sequential([
        Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01), input_shape=(X_train.shape[1],)),
        Dropout(0.25),
        Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
        Dropout(0.25),
        Dense(1, activation='sigmoid')  # for binary classification
    ])

    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# cross validation and pick model having the highest f1 score.
kf = KFold(n_splits=5, shuffle=True, random_state=seed_value)
f1_scores = []
best_models = []

for train_index, val_index in kf.split(X_train):
    X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[val_index]
    y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[val_index]

    model = create_model()

    # define early stopping call back
    early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

    # define ModelCheckpoint call back
    model_checkpoint = ModelCheckpoint('best_model_weights.h5', monitor='val_loss', save_best_only=True, save_weights_only=True)

    # define learning rate scheduler
    lr_scheduler = LearningRateScheduler(warmup)

    # train the model
    history = model.fit(X_train_fold, y_train_fold, epochs=100, batch_size=128,
                        validation_data=(X_val_fold, y_val_fold),
                        callbacks=[early_stopping, model_checkpoint, lr_scheduler], verbose=0)
```

```
        # train the model
        history = model.fit(X_train_fold, y_train_fold, epochs=100, batch_size=128,
                            validation_data=(X_val_fold, y_val_fold),
                            callbacks=[early_stopping, model_checkpoint, lr_scheduler], verbose=0)

        # to looking for the optimal epoch
        best_epoch = np.argmin(history.history['val_loss']) + 1

        # model evaluation at the optimal epoch
        model.load_weights('best_model_weights.h5')

        y_pred_prob = model.predict(X_val_fold)
        y_pred = (y_pred_prob > 0.5).astype(int).ravel()  # 예측 결과를 1차원 배열로 변환
        y_val_fold = y_val_fold.ravel()  # 실제 값을 1차원 배열로 변환
        f1 = f1_score(y_val_fold, y_pred)

        f1_scores.append(f1)
        best_models.append(model)

    best_model_index = np.argmax(f1_scores)
    mean_f1_score = np.mean(f1_scores)
    best_model = best_models[best_model_index]

    # select the best model based on the average score of cross validation score by using K-fold
    selected_models = {"Neuron Network": best_model}

    print(f'Best F1 Score: {f1_scores[best_model_index]}')
```

6.4. SVM

```
[54]  1 # Support Vector Machine (SVM) - L2 regularization is built-in
      2 from sklearn.svm import SVC
      3 from sklearn.pipeline import Pipeline
      4
      5 model = SVC(kernel='rbf', C=1.0)  # An SVM model using the RBF kernel, where C is the regularization parameter
      6
      7 model.fit(X_train, y_train)
      8
      9 models["SVM"]=model
     10
     11 y_pred = grid_search.predict(X_test)
     12 y_pred = (y_pred > 0.5).astype(int)
     13
     14 print("Accuracy:", accuracy_score(y_test, y_pred))
     15 print("F1:", f1_score(y_test, y_pred))
     16 print("ROC AUC:", roc_auc_score(y_test, y_pred))
```

6.5 Polynomial

```python
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

pipeline = Pipeline([
    ('poly_features', PolynomialFeatures()),
    ('linear_regression', LinearRegression())
])


param_grid = {
    'poly_features__degree': [2, 3, 4],
    'linear_regression__fit_intercept': [True, False]
}


grid_search = GridSearchCV(pipeline, param_grid, cv=kf, scoring='neg_mean_squared_error', refit=True, n_jobs=-1)
grid_search.fit(X_train, y_train)


print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(-grid_search.best_score_))

model_selected = grid_search.best_estimator_

y_pred = model_selected.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_prob[:, 1]))
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))

```

6.7 Perceptron

```python
from sklearn.linear_model import Perceptron

model = Pipeline([
    ('scaler', StandardScaler()),
    ('perceptron', Perceptron(random_state=random_state))
])

# Define the hyperparameters and their possible values
param_grid = {
    'perceptron__penalty': [None, 'l2', 'l1', 'elasticnet'],
    'perceptron__alpha': [0.0001, 0.001, 0.01, 0.1],
    'perceptron__max_iter': [1000, 2000, 3000],
    'perceptron__tol': [1e-3, 1e-4, 1e-5]
}

# Grid search with cross-validation
grid_search = GridSearchCV(model, param_grid, cv=kf, scoring=scoring, refit=True, n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best parameters: ", grid_search.best_params_)
print("Best CV score: {:.6f}".format(grid_search.best_score_))

# Save the best estimator to the models dictionary
models["Perceptron"] = grid_search.best_estimator_

# Make predictions
y_pred = grid_search.best_estimator_.predict(X_test)

# Evaluate the model
print("")
print("Accuracy:", accuracy_score(y_test, y_pred))
print("F1:", f1_score(y_test, y_pred, average='weighted'))
```