CS3111 – Introduction to Machine Learning

Lab 01 – Feature Engineering

210173T – Gallage D.

Lab Report

Data preprocessing is a crucial step in the data analysis pipeline that involves transforming raw data into a clean and structured format suitable for further analysis and modeling. This preparatory phase plays a fundamental role in enhancing the quality of data by addressing various issues such as missing values, outliers, and inconsistencies. By performing data preprocessing techniques, such as data cleaning, feature scaling, feature engineering, and handling categorical variables, analysts can ensure that the data is accurate, complete, and ready for use in analytical models. Effective data preprocessing lays the foundation for meaningful insights and reliable predictions, contributing significantly to the success of data-driven decision-making processes.

In this lab, we use the following data preprocessing techniques to prepare the data for analysis and modeling tasks.

- Data cleaning
- Handling missing values
- Imputing values for outliers
- Feature encoding
- Handling categorical variables
- Dimensionality reduction
- Feature selection

1. Data Cleaning

Data cleaning is an essential component of the data preprocessing pipeline that focuses on identifying and rectifying errors, inconsistencies, and inaccuracies within the dataset. It involves a series of steps aimed at ensuring the quality and integrity of the data by addressing issues such as missing values, duplicate entries, outliers, and noise. Through meticulous data cleaning techniques, analysts aim to improve the reliability and accuracy of the dataset, thus enhancing the effectiveness of subsequent analysis and modeling tasks. By detecting and rectifying errors early in the data processing workflow, data cleaning helps mitigate the risk of biased insights and erroneous conclusions, ultimately leading to more robust and reliable decision-making based on the data. In this lab, we delve into various data cleaning techniques and strategies to effectively prepare the data for further analysis and modeling endeavors.

As a first step in data cleaning, we remove columns with many missing values, using a threshold of 50%. This helps us focus on variables with enough information and ensures our analysis is based on reliable data. By removing columns with a high percentage of missing values, we improve the quality of our dataset for further analysis and modeling.

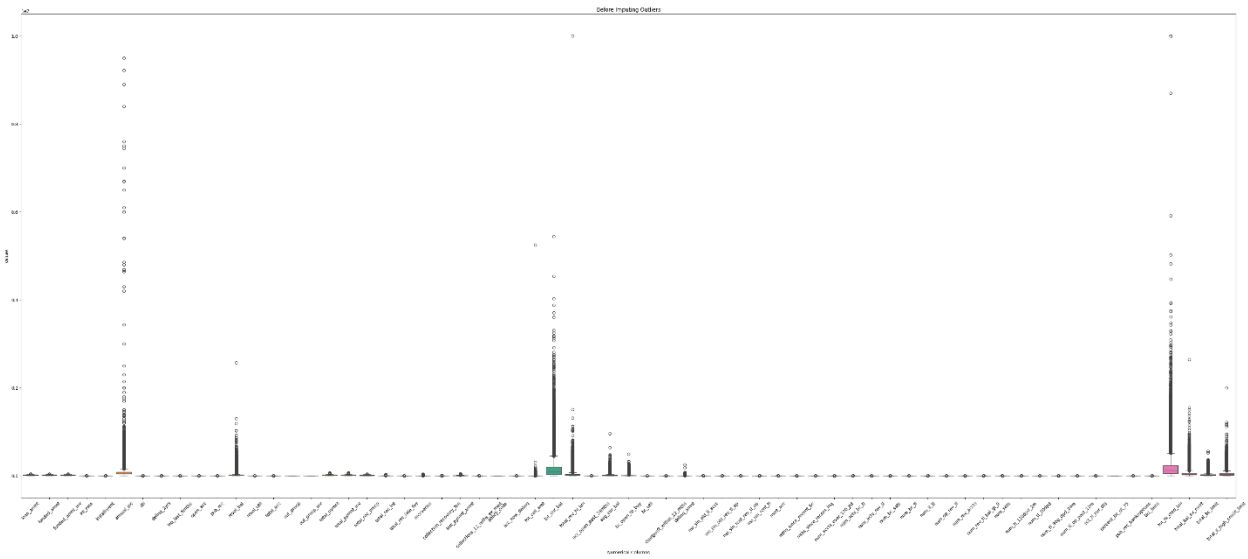
```
# Remove columns with more than 50% missing values
threshold = 0.5
columns_to_drop = X_temp.columns[X_temp.isnull().mean() > threshold].tolist()

X_temp.drop(columns_to_drop, axis=1, inplace=True)
```

After removing columns with many missing values, the next step in data cleaning involves imputing missing values with new values. For categorical columns, we replace missing values with the mode (most frequent value), while for numerical columns, we use the mean value. This process ensures that the dataset is complete and ready for analysis, while maintaining the integrity of the data. Imputing missing values with appropriate replacements helps preserve the overall structure and distribution of the dataset, enabling accurate analysis and modeling.

```
# Imput missing values in the columns
for col in X_temp.columns:
    if X_temp[col].dtype == 'object':
        mode_value = X_temp[col].mode()[0]
        X_temp[col] = X_temp[col].fillna(mode_value)
    else:
        mean_value = X_temp[col].mean()
        X_temp[col] = X_temp[col].fillna(mean_value)
```

After imputing missing values, the next step in data cleaning involves handling outliers. Outliers are data points that significantly deviate from the rest of the dataset and can skew analysis results. To address outliers, we identify them using a method that compares each value to the median of the column.

Once identified, outliers are imputed with the median value of the column. This process ensures that extreme values do not unduly influence analysis outcomes, leading to more robust and reliable results. By handling outliers effectively, we ensure that our dataset is better suited for subsequent analysis and modeling tasks.




```
# Define a function to replace outliers with the median
def replace_outliers(series):
    series_temp = series.copy()  # Create a copy of the input series
    median = series_temp.median()
    std_deviation = series_temp.std()
     outliers = (series_temp - median).abs() > 5 * std_deviation  # Adjust the
threshold as needed
    series_temp[outliers] = median
    return series_temp

# Impute outliers in numerical columns
for col in numerical_columns:
    X_temp[col] = replace_outliers(X_temp[col])
```

After handling outliers, the next step in data cleaning involves removing columns with constant values throughout. These columns, also known as constant features, do not provide any meaningful information for analysis or modeling as they contain the same value for all records. By removing such columns, we streamline the dataset and focus our analysis on variables that exhibit variability and contribute to the predictive power of the model. This process helps improve the efficiency of subsequent analysis and modeling tasks by reducing the dimensionality of the dataset and eliminating redundant information. Additionally, removing constant features helps prevent potential issues such as overfitting, where the model learns noise rather than meaningful patterns in the data.

```
# Get the number of unique values in each column
unique_counts = X_temp.nunique()

# Filter columns with only one unique value
constant_columns = unique_counts[unique_counts == 1].index.tolist()

X_temp.drop(constant_columns, axis=1, inplace=True)
```

2. Feature Encoding

Feature encoding is a crucial technique in data preprocessing that involves converting categorical variables into a numerical format suitable for machine learning algorithms. Categorical variables represent qualitative data, such as gender or color, which cannot be directly used in most machine learning models. Feature encoding enables us to transform these categorical variables into numerical representations that algorithms can understand and process. This process is essential for effectively leveraging categorical data in machine learning models, ensuring that they can learn from all relevant information in the dataset. Feature encoding techniques include one-hot encoding, label encoding, and ordinal encoding, each suited for different types of categorical variables and modeling scenarios. By applying feature encoding, we enable machine learning models to effectively incorporate categorical variables into their decision-making processes, ultimately improving their predictive performance and accuracy.

I used the scikit-learn OrdinalEncoder to transform ordinal categorical variables into numerical representations. This allowed me to maintain the ordinal relationship between the categories while preparing the data for machine learning models. The OrdinalEncoder assigned unique integer values to each category, preserving their order or hierarchy. By incorporating this feature encoding technique into my data preprocessing workflow, I was able to enhance the predictive accuracy of my machine learning models by effectively leveraging the ordinal nature of the data.

```
# Perform ordinal encoding on ordinal columns
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()

X_temp[ordinal_cols] = ordinal_encoder.fit_transform(X_temp[ordinal_cols])

# Convert encoded values to integers
X_temp[ordinal_cols] = X_temp[ordinal_cols].astype(int)
```

I used the scikit-learn LabelEncoder to convert categorical nominal variables into numerical representations. This encoder assigned a unique integer value to each category, providing a numerical label for each distinct value in the column. By using the LabelEncoder, I transformed categorical nominal variables into a format that machine learning models could interpret and process effectively. This preprocessing step enabled me to incorporate categorical nominal data into my machine learning models, improving their predictive performance by converting qualitative data into a numerical format.

```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()

for col in nominal_cols:
    X_temp[col] = label_encoder.fit_transform(X_temp[col])
```

To encode columns with date values, I followed this strategy:

I first found the minimum date among all the values in the column. Then, I converted each date value to a numerical representation by calculating the number of days elapsed from the minimum date. This approach transformed date columns into a format that machine learning models could understand, allowing me to incorporate date information into my models for improved predictive performance.

```
# Set initial minimum date
min_date = pd.to_datetime('2024-01-01')

for col in date_cols:
    X_temp[col] = pd.to_datetime(X_temp[col], format='%b-%Y')
    if X_temp[col].min() < min_date:
        min_date = X_temp[col].min()

print(f"\nmin_date: {min_date}\n")

# Set timestamps for date columns
for col in date_cols:
    X_temp[col] = (X_temp[col] - min_date).dt.days.astype(int)
```

I used the TargetEncoder from the categorical_encoders library to encode columns that have many unique values. Target encoding is a feature encoding technique that assigns each category in a categorical variable a numerical value based on the target variable's mean or other statistical measures. This encoding method is particularly useful for columns with a large number of unique categories, as it captures the relationship between the categorical variable and the target variable, potentially improving the predictive power of the model. By incorporating target encoding into my data preprocessing pipeline, I transformed categorical variables into numerical representations that could be effectively utilized by machine learning models for better predictive accuracy.

```
from category_encoders import TargetEncoder

# Perform target encoding on 'emp_title' and 'title' columns
target_encoder = TargetEncoder()
X_temp[many_unique_cols] = target_encoder.fit_transform(X_temp[many_unique_cols],
y_temp)
```

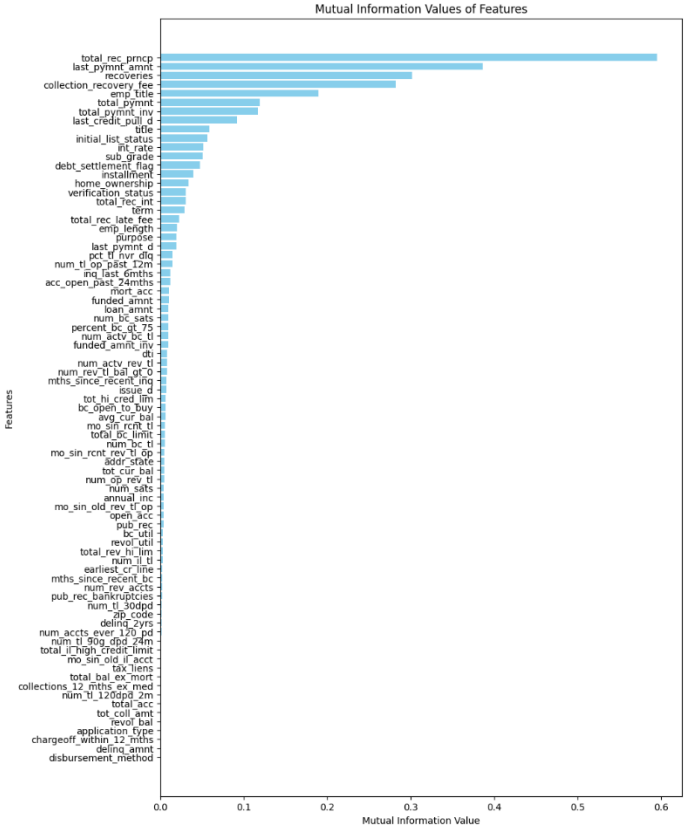
3. Feature Selection

Feature selection is a crucial aspect of the feature engineering process that involves identifying and selecting the most relevant and informative features from the dataset for use in machine learning models. The goal of feature selection is to improve model performance by reducing the dimensionality of the dataset and focusing on the most important features that contribute to predictive accuracy. This process helps mitigate the curse of dimensionality, reduce overfitting, and improve model interpretability by eliminating redundant or irrelevant features. Feature selection techniques include filter methods, wrapper methods, and embedded methods, each employing different strategies to identify and select the most relevant features based on criteria such as feature importance, correlation, or predictive power. By performing feature selection, data scientists and machine learning practitioners can streamline the modeling process, enhance model performance, and gain deeper insights into the underlying patterns within the data.

Mutual information is a statistical metric that measures the amount of information obtained about one variable through the knowledge of another variable. In the context of feature selection, mutual information quantifies the mutual dependence between each feature and the target variable. It assesses the degree of association or dependency between a feature and the target variable, irrespective of the linear relationship.

To select features using mutual information, I computed the mutual information score between each feature and the target variable. This score indicates how much information each feature provides about the target variable. Features with high mutual information scores are considered informative and are retained for further analysis, while features with low scores are deemed less relevant and are excluded from the analysis. By selecting features based on mutual information, I focused on identifying the most informative features that have a significant association with the target variable, thus improving the predictive performance of the machine learning model.

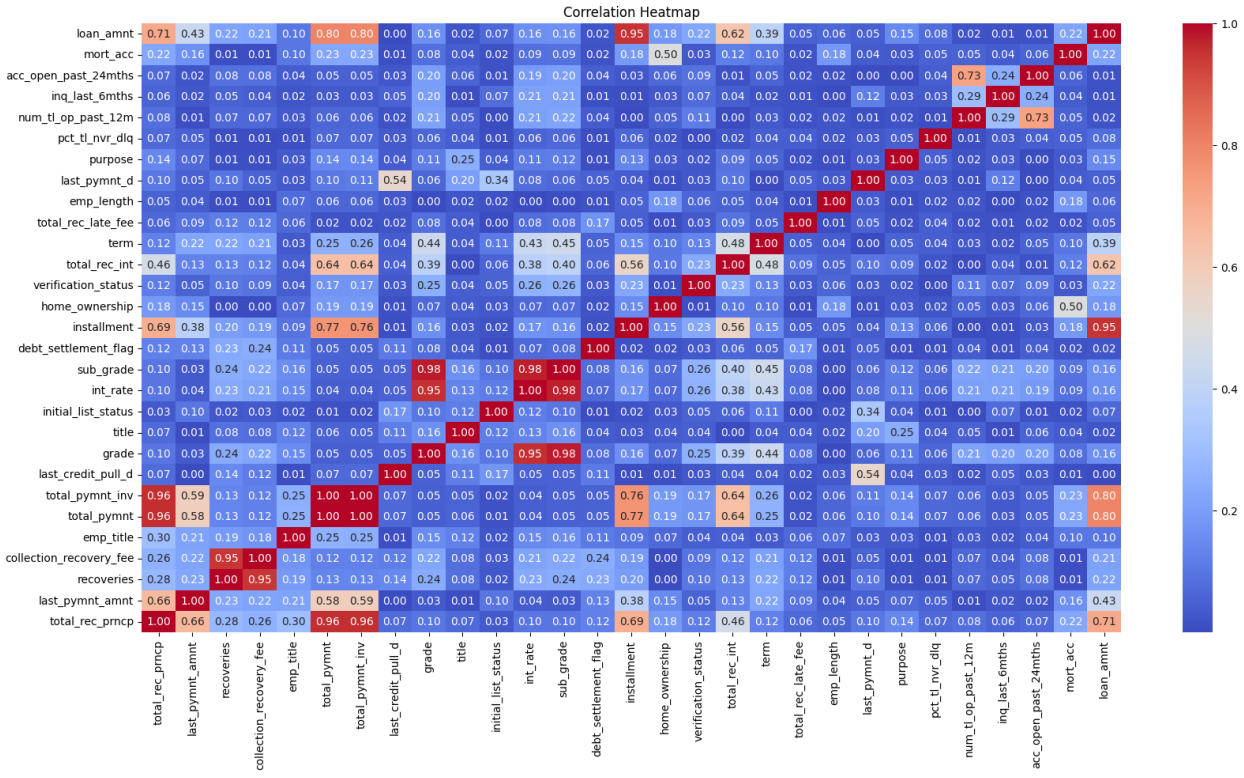
```
from sklearn.feature_selection import mutual_info_classif

# Compute mutual information for each feature
mi_values = mutual_info_classif(X_temp, y_temp, discrete_features='auto',
random_state=42)

# Create a DataFrame to store feature names and their corresponding MI values
mi_df = pd.DataFrame({'Feature': X_temp.columns, 'MI Value': mi_values})
mi_df = mi_df.sort_values(by='MI Value', ascending=False) # Descending order

threshold_value = 0.01
selected_features = mi_df[mi_df['MI Value'] > threshold_value]['Feature'].tolist()
```



After selecting features using mutual information, the next step involved calculating the correlation between features and removing features with high correlation.

Correlation is a statistical measure that quantifies the strength and direction of the linear relationship between two variables. High correlation between features indicates redundancy, as these features convey similar information to the model, potentially leading to multicollinearity issues.

To identify features with high correlation, I computed the correlation matrix between all pairs of selected features. Features with correlation coefficients above a certain threshold (e.g., 0.8 or -0.8) were considered highly correlated and were subsequently removed from the dataset.
By removing features with high correlation, I aimed to reduce redundancy in the dataset and improve the stability and interpretability of the machine learning model. This step helped prevent multicollinearity issues and ensured that the remaining features provided unique and complementary information to the model, ultimately enhancing its predictive performance.



After applying feature selection techniques, we successfully reduced the dataset from 144 features to 23 features.

Subsequently, we performed encoding for the selected features across all datasets including train, validation, and test sets to train the model. It's important to note that all transformations applied to the validation and test datasets were based on the transformations applied to the train dataset. This ensures consistency in the preprocessing steps and prevents data leakage, ultimately ensuring the model's reliability and generalizability. By encoding the selected features and applying consistent transformations across all datasets, we prepared the data for training the machine learning model, which will then be evaluated and tested for its predictive performance.

After preprocessing the data, I trained an XGBoost classifier model and performed hyperparameter tuning using cross-validation to enhance its accuracy. The hyperparameters I tuned were 'learning_rate', 'max_depth', and 'n_estimators'.

- learning_rate: 0.3, 0.4, 0.5, 0.6
- max_depth: 2, 3, 4, 5
- n_estimators: 100, 200, 300, 400

The grid search resulted in finding the best model with the following hyperparameters:

- learning_rate: 0.5
- max_depth: 4
- n_estimators: 400

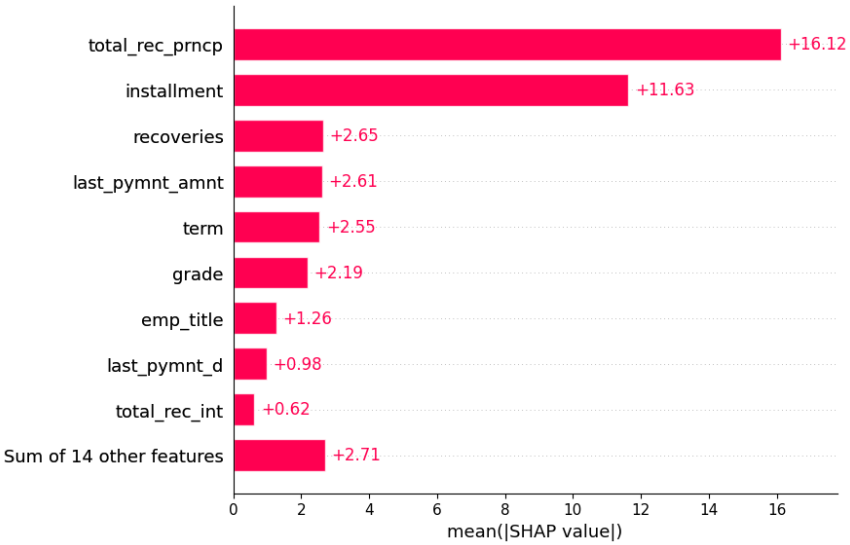
```
from sklearn.model_selection import GridSearchCV
from xgboost import XGBClassifier

param_grid = {
    'learning_rate': [0.3, 0.4, 0.5, 0.6],
    'max_depth': [2, 3, 4, 5],
    'n_estimators': [100, 200, 300, 400],
}

xgb_model = XGBClassifier()

# Create GridSearchCV instance
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=3,
scoring='accuracy')
grid_search.fit(X_train, y_train)

print("Best hyperparameters:", grid_search.best_params_)

best_xgb_model = grid_search.best_estimator_
```

To validate the model's performance, I used the validation dataset and obtained an accuracy score of 0.99878. Based on this score, I concluded that the model is neither overfitted nor underfitted. An accuracy score close to 1 indicates that the model is performing well on unseen data and generalizing effectively to new instances.

Overall, the trained XGBoost classifier with the optimized hyperparameters demonstrated strong performance on the validation dataset, suggesting that it is adequately capturing the underlying patterns in the data without exhibiting significant signs of overfitting or underfitting.

SHAP Analysis

SHAP (SHapley Additive exPlanations) values provide insights into the contribution of each feature to the model's predictions. The absolute SHAP mean plot visualizes the average absolute SHAP values for each feature, showcasing the importance of each feature in determining the model's predictions. By examining the absolute SHAP mean plot, we can quickly identify the most influential features in the model and gain insights into how they impact the predictions. This allows us to assess the overall performance and behavior of the model, understand its strengths and weaknesses, and identify areas for improvement.



For a detailed exploration of the code and analysis conducted in this report, please follow this link to access the Jupyter Notebook:

- Jupyter notebook of the feature engineering lab