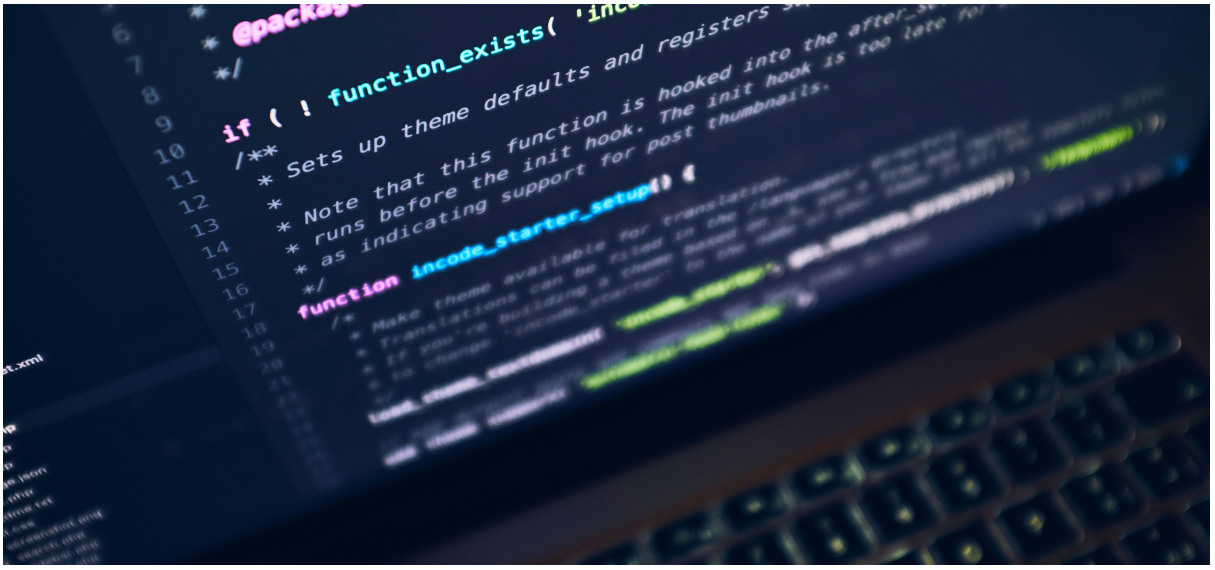

MSME Credit Exposure Prediction model

Predicting credit exposure is a critical task for financial institutions as it helps in assessing potential risks associated with lending activities. The goal of this project is to develop a robust model that can predict the credit exposure of a client based on historical data and various financial indicators. The dataset used for this analysis contains numerous features, including client demographics, financial history, transaction details, and other relevant attributes. Understanding and processing this data accurately is crucial to developing a predictive model that can generalize well to unseen data.
In this project, we will explore multiple machine learning models, including ensemble methods, to identify the best-performing model. We'll begin by loading and exploring the dataset, followed by feature selection, model training, and evaluation. The results will be compared using various performance metrics to determine the most accurate model.

The code is as follows:-

1. Loading the dataset

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxSca
```

```
ler

# Load the dataset from the specified file path
file_path = "D:/RI IIM INDR Files/Market report Cred Exp_fi
lled.csv"
df = pd.read_csv(file_path)

# Display first few rows of the dataset
print("First few rows of the dataset:")
print(df.head())

# summary of dataset to check data types and missing values
print("\nDataset information:")
print(df.info())

# Check missing values
print("\nMissing values in each column:")
print(df.isnull().sum())

# Remove columns with all missing values
df.dropna(axis=1, how='all', inplace=True)

# Print column names to identify correct target column
print("\nColumn names in the dataset:")
print(df.columns)

# Data Cleaning

# Select numerical columns
numerical_cols = df.select_dtypes(include=['number']).colum
ns

# Fill missing values with the mean for numerical columns
df[numerical_cols] = df[numerical_cols].fillna(df[numerical
_cols].mean())

# Drop rows with missing values
df.dropna(inplace=True)
```

```
# Convert data types if not done already
for col in df.columns:
    if df[col].dtype == 'object':  # Convert object columns
to categorical if necessary
        df[col] = df[col].astype('category').cat.codes

# Print first few rows after cleaning
print("\nFirst few rows after cleaning:")
print(df.head())

# Specify the actual target column
target_column = 'Tot_CredExp'

# Check if the target column is present
if target_column not in df.columns:
    print(f"\nError: Target column '{target_column}' not fo
und in the dataset. Please update the target_column variabl
e.")
else:
    # Identify features and target
    features = df.drop(columns=target_column)
    target = df[target_column]

    # Normalize/Scale data
    # 1. Standardization
    scaler_standard = StandardScaler()
    features_standardized = scaler_standard.fit_transform(f
eatures)

    # 2. Min-Max Scaling
    scaler_minmax = MinMaxScaler()
    features_minmax_scaled = scaler_minmax.fit_transform(fe
atures)

    print("\nFirst few rows of standardized features:")
    print(pd.DataFrame(features_standardized, columns=featu
res.columns).head())
```

```
    # display min-max scaled features
    print("\nFirst few rows of Min-Max scaled features:")
    print(pd.DataFrame(features_minmax_scaled, columns=feat
ures.columns).head())
```

`StandardScaler` and `MinMaxScaler` from `sklearn.preprocessing` are imported for scaling the features in the dataset. `StandardScaler` standardizes features by removing the mean and scaling to unit variance, while `MinMaxScaler` scales features to a given range, typically [0, 1].

Q. What is scaling?

Ans. Scaling is an essential step in data preprocessing, especially when the data is going to be used in machine learning models. It is a process of transforming the numerical features of the dataset so that they fit within a specific range or distribution.

Example; for Min-Max scaling, the formula used is

Formula:

$$x' = \frac{(x - \min)}{(\max - \min)}$$

Where:

- x is the original value,
- $\min$ and $\max$ are the minimum and maximum values of the feature,
- x' is the scaled value.

`df.head()` displays the first five rows of the dataset, giving you a preview of the data. The output will include all columns in the dataset with their first five values.

`df.info()` provides a summary of the dataset, including the number of non-null

entries, data types of each column, and memory usage.

`df.isnull().sum()` counts the number of missing values in each column.

`df.dropna(axis=1, how='all')` drops any columns that have all values missing.

`df.columns` prints the names of all columns in the dataset.

`df.select_dtypes(include=['number'])` selects only the numerical columns from the dataset, storing their names in `numerical_cols`.

`df[numerical_cols].fillna(df[numerical_cols].mean())` fills missing values in the numerical columns with the mean of each respective column.

`df.dropna(inplace=True)` removes any remaining rows that still contain missing values

For this model, we have chosen `Tot_CredExp` as the target column. `StandardScaler()` standardizes the features, ensuring they have a mean of 0 and a standard deviation of 1.

`features_standardized` stores the standardized feature values.

`MinMaxScaler()` scales features to a specified range, typically [0, 1].

2. Initializing a correlation matrix

```
correlation_matrix = df.corr()
print(correlation_matrix['Tot_CredExp'].sort_values(ascendi
ng=False))
```

Q. What is a correlation matrix?

Ans. A correlation matrix is a table showing correlation coefficients between variables. Each cell in the matrix represents the correlation between two variables. The correlation coefficient values range from -1 to +1, where:

1. **+1** indicates a perfect positive correlation (as one variable increases, the other also increases).
2. **-1** indicates a perfect negative correlation (as one variable increases, the other decreases).
3. **0** indicates no correlation (the variables do not have any linear relationship).

`df.corr()` calculates the Pearson correlation coefficient for each pair of numerical columns in the DataFrame `df`.

3. Chi-square calculation for features

```
from sklearn.feature_selection import chi2
from sklearn.preprocessing import LabelEncoder

# Ensure features are non-negative
features_non_negative = features - features.min().min() + 1

# Encode the target variable
label_encoder = LabelEncoder()
encoded_target = label_encoder.fit_transform(df['Tot_CredEx
p'])

# Apply the Chi-square test
chi2_scores, p_values = chi2(features_non_negative, encoded
_target)

# Display the Chi-square scores, sorted in descending order
print(pd.Series(chi2_scores, index=features.columns).sort_v
alues(ascending=False))
```

`chi2` **from** `sklearn.feature_selection`**:** This function is used to compute the Chi-square statistic between each feature and the target variable. It's commonly used in feature selection, especially for classification tasks.

`LabelEncoder` **from** `sklearn.preprocessing`**:** This is used to convert categorical labels into numerical values. Since Chi-square tests are typically used for categorical data, this step is necessary if the target variable is not already numerical.

The Chi-square test requires that the features (input variables) be non-negative, as it is based on frequency counts.

`LabelEncoder` converts `Tot_CredExp` labels into a format that can be processed by the Chi-square test. `fit_transform` converts the target variable into integer labels. For instance, if `Tot_CredExp` has categories like "Low," "Medium," and "High," it might be encoded as 0, 1, and 2, respectively.

For each feature, the Chi-square statistic is calculated to measure how strongly the feature is associated with the target variable. A higher Chi-square score indicates a stronger association.

4. Initializing the Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
import pandas as pd

model = RandomForestRegressor()
model.fit(features, target)

importance = model.feature_importances_
feature_importance = pd.Series(importance, index=features.c
olumns).sort_values(ascending=False)

print("Feature Importances:")
print(feature_importance)
```

`RandomForestRegressor` **from** `sklearn.ensemble`**:** This is a machine learning model that is part of the ensemble learning methods. It builds multiple decision trees and merges them to get a more accurate and stable prediction.

`model = RandomForestRegressor()` creates an instance of the `RandomForestRegressor` model. By default, it uses 100 decision trees (specified by the `n_estimators` parameter)
The model learns the relationships between the input features and the target

variable by creating multiple decision trees and combining their results to improve accuracy and reduce overfitting.

`model.feature_importances_` is an attribute of the trained Random Forest model that stores the importance scores for each feature. These scores indicate the relative importance of each feature in predicting the target variable.

- **Model Interpretation:** Understanding feature importance helps in interpreting the model's decisions, which is crucial in making a model more transparent and explainable.
- **Feature Selection:** By identifying the most important features, one can reduce the dimensionality of the data, improve model performance, and reduce overfitting by focusing on the most relevant features.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import RFE

model = RandomForestRegressor()

# Initialize RFE with model and no. of features to select
rfe = RFE(model, n_features_to_select=5)
fit = rfe.fit(features, target)

# print selected features
print("Selected features:", fit.support_)
print("Feature ranking:", fit.ranking_)
```

`RandomForestRegressor` **from** `sklearn.ensemble` is a machine learning model used for regression tasks. It builds multiple decision trees and averages their predictions to improve accuracy.
Recursive Feature Elimination (RFE) is a feature selection method that recursively removes the least important features based on the performance of the model.

Number of features selected in this regression model=5 (`n_features_to_select=5`).

`rfe.fit(features, target)` applies the RFE method to the given `features` and `target`. It trains the Random Forest model and iteratively removes the least

significant features according to the model's performance, eventually selecting the top 5 features.

`fit.support_`: This attribute provides a boolean array where `True` indicates that the feature is selected, and `False` means it is not. It shows which features have been chosen by RFE.

`fit.ranking_`: This attribute provides an array of rankings for all features, where the rank indicates the importance of each feature. Features with a rank of 1 are the most important and selected features, while features with higher ranks are less important.

To learn more about Random Forest Regression, one can visit Geeks for Geeks or Scikit Learn.

```
from sklearn.feature_selection import SelectKBest, f_regres
sion

selector = SelectKBest(score_func=f_regression, k='all')
fit = selector.fit(features, target)
scores = pd.Series(fit.scores_, index=features.columns)
print(scores.sort_values(ascending=False))
```

`SelectKBest` **from** `sklearn.feature_selection`: This is a feature selection method that selects the top `k` features based on a statistical score.

`f_regression` **from** `sklearn.feature_selection`: This is a scoring function specifically designed for regression tasks. It computes the F-statistic between each feature and the target variable, measuring the linear relationship between them.

`score_func=f_regression`: Specifies that the `f_regression` scoring function should be used to evaluate the importance of each feature. This function computes the F-statistic, which tests the linear dependency between each feature and the target variable.

`k='all'`: Indicates that all features should be kept and scored. Normally, `k` can

be set to a specific number to select the top `k` features. Here, by setting `k='all'`, we ensure that scores are calculated for all features, and no feature is discarded.

5. Feature Engineering

```
from sklearn.feature_selection import SelectKBest, f_classi
f
import pandas as pd

# Example DataFrame df initialization (make sure to replace
this with your actual DataFrame loading code)
# df = pd.read_csv('your_file.csv') # Replace with actual d
ata loading method

# Check for missing values
print("Missing values in df:\n", df.isna().sum())

# Fill or drop missing values as needed
df = df.fillna(0)  # This fills NaNs with 0; you might need
a different approach depending on your data

# Check if required columns exist
if 'NPA_amount' in df.columns and 'Tot_CredExp_SMLgE' in d
f.columns:
    # Feature engineering: Calculate Growth Rate
    df['Growth_Rate'] = (df['Tot_CredExp'] / df['Tot_CredEx
p_SMLgE'] - 1) * 100
    print("Growth Rate calculated successfully.")

    # Feature engineering: Calculate Delinquency Rate
    df['Del_Rate'] = df['NPA_amount'] / df['Tot_CredExp']
    print("Delinquency Rate calculated successfully.")
else:
    print("Required columns are missing.")

# Define features and target
```

```
features = df[['NPA_amount', 'Tot_CredExp_SMLgE', 'VintDe
l']]
target = df['Tot_CredExp']

# Handle missing values in features and target
features = features.fillna(0)  # This fills NaNs with 0; ad
just as needed
target = target.fillna(0)      # This fills NaNs with 0; ad
just as needed

# Feature selection
selector = SelectKBest(score_func=f_classif, k='all')
fit = selector.fit(features, target)
scores = pd.Series(fit.scores_, index=features.columns)
print(scores.sort_values(ascending=False))
```

```
#print with new columns
print(df)
print(df[['Tot_CredExp', 'VintDel', 'Growth_Rate', 'Del_Rat
e']])
print(df[['Tot_CredExp', 'VintDel', 'Growth_Rate', 'Del_Rat
e']].head())
print(df['Growth_Rate'].describe())
print(df['Del_Rate'].describe())
df.to_csv("report_with_rate.csv", index=False)
```

- **Missing Values Handling:**
  - Checks for missing values in the DataFrame `df`.
  - Fills missing values with `0`.
- **Feature Engineering:**
  - Calculates `Growth_Rate` using the formula `((Tot_CredExp / Tot_CredExp_SMLgE) - 1) * 100`.
  - Calculates `Del_Rate` as the ratio of `NPA_amount` to `Tot_CredExp`.
  - Both calculations are contingent on the presence of the necessary columns.

- **Feature and Target Definition:**
  - Defines `features` as a subset of columns (`NPA_amount`, `Tot_CredExp_SMLgE`, `VintDel`).
  - Defines `target` as the `Tot_CredExp` column.
- **Missing Values Handling in Features and Target:**
  - Fills any remaining missing values in `features` and `target` with `0`.
- **Feature Selection:**
  - Uses `SelectKBest` with the `f_classif` scoring function to select and score features.
  - Outputs the scores for each feature, sorted in descending order.

6. Principal Component Analysis

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)  # Reduce to 2 dimensions
principal_components = pca.fit_transform(features)
pca_df = pd.DataFrame(data=principal_components, columns=
['PC1', 'PC2'])


import pandas as pd
import numpy as np
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import umap
features = df.drop(columns=['Tot_CredExp']) #our target col
umn

# check no. of samples
num_samples = features.shape[0]
print("Number of samples:", num_samples)

# adjust perplexity or use PCA for dimensionality reduction
```

```
before t-SNE
if num_samples > 30:
    tsne = TSNE(n_components=2, perplexity=min(30, num_samp
les - 1))
    tsne_results = tsne.fit_transform(features)
    tsne_df = pd.DataFrame(data=tsne_results, columns=['TSN
E1', 'TSNE2'])
else:
    # Use UMAP if t-SNE is not feasible
    umap_model = umap.UMAP(n_components=2)
    umap_results = umap_model.fit_transform(features)
    tsne_df = pd.DataFrame(data=umap_results, columns=['UMA
P1', 'UMAP2'])

print(tsne_df.head())
```

- **PCA (Principal Component Analysis):**
  - **Objective:** Reduce the dimensionality of `features` to 2 components.
  - **Output:** Transforms `features` into `principal_components`, creating a DataFrame with columns `PC1` and `PC2`.
- **t-SNE (t-Distributed Stochastic Neighbor Embedding):**
  - **Objective:** Perform dimensionality reduction to 2D for visualization.
  - **Condition:**
    - If the number of samples (`num_samples`) is greater than 30, t-SNE is applied with an adjusted `perplexity` parameter.
    - If fewer than 30 samples, UMAP is used instead of t-SNE.
  - **Output:**
    - For t-SNE, creates `tsne_df` with columns `TSNE1` and `TSNE2`.
    - For UMAP, creates `tsne_df` with columns `UMAP1` and `UMAP2`.

7. Training the Random Forest Regressor Model

```
#split
import pandas as pd
from sklearn.model_selection import train_test_split
features = df.drop(columns=['Tot_CredExp'])
target = df['Tot_CredExp']

# split data into training and test sets (85% training, 15%
test)
X_train, X_temp, y_train, y_temp = train_test_split(feature
s, target, test_size=0.15, random_state=42)

# split temporary set into validation and test sets (50% va
lidation, 50% test of the 15%)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_t
emp, test_size=0.5, random_state=42)

print("Training set size:", X_train.shape)
print("Validation set size:", X_val.shape)
print("Test set size:", X_test.shape)
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

from sklearn.ensemble import RandomForestRegressor

# Initialize the model
model = RandomForestRegressor(random_state=42)

# Train the model
model.fit(X_train_scaled, y_train)
```

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20]
}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)
best_model = grid_search.best_estimator_

from sklearn.metrics import mean_absolute_error, mean_squar
ed_error, r2_score

# Predictions
y_val_pred = best_model.predict(X_val_scaled)
y_test_pred = best_model.predict(X_test_scaled)

# Evaluation metrics
print("Validation MAE:", mean_absolute_error(y_val, y_val_p
red))
print("Validation MSE:", mean_squared_error(y_val, y_val_pr
ed))
print("Validation R^2:", r2_score(y_val, y_val_pred))

print("Test MAE:", mean_absolute_error(y_test, y_test_pre
d))
print("Test MSE:", mean_squared_error(y_test, y_test_pred))
print("Test R^2:", r2_score(y_test, y_test_pred))
```

Data Splitting:

- **Step 1:** Splits the data into training (85%) and temporary (15%) sets using `train_test_split`.
- **Step 2:** Further splits the temporary set into validation (7.5%) and test (7.5%) sets.
- **Resulting Sets:**
  - `X_train` and `y_train`: Training set features and target.

- `X_val` and `y_val`: Validation set features and target.
- `X_test` and `y_test`: Test set features and target.

Standard Scaling:

- **Objective:** Normalize the features in the training, validation, and test sets.
- **Steps:**
  - `X_train` is scaled using `StandardScaler`'s `fit_transform` method, which learns the scaling parameters from the training data and applies the transformation.
  - `X_val` and `X_test` are scaled using the `transform` method, applying the same parameters learned from the training data.

- **Model Initialization and Training:**
  - **Model:** A `RandomForestRegressor` is initialized with a fixed `random_state` for reproducibility.
  - **Training:** The model is trained on the scaled training set (`X_train_scaled`, `y_train`).

- **Grid Search for Hyperparameter Tuning:**
  - **Objective:** Optimize the `RandomForestRegressor` by finding the best combination of hyperparameters.
  - **Parameters Tuned:**
    - `n_estimators`: Number of trees in the forest (100, 200).
    - `max_depth`: Maximum depth of each tree (10, 20).
  - **Cross-Validation:** A 5-fold cross-validation (`cv=5`) is used to evaluate the performance of each combination.
  - **Result:** The best model is selected based on cross-validation performance.
- **Model Evaluation:**
  - **Predictions:**

- `y_val_pred`: Predictions on the validation set.
- `y_test_pred`: Predictions on the test set.

- **Evaluation Metrics:**
  - **Mean Absolute Error (MAE):** Measures the average magnitude of errors in predictions.
  - **Mean Squared Error (MSE):** Measures the average of the squares of errors.
  - **R^2 Score:** Indicates the proportion of variance in the target variable explained by the model.

8. Exploratory Data Analysis

```
import matplotlib.pyplot as plt
import seaborn as sns


#EDA
# Plot histogram for each feature
df.hist(figsize=(12, 10), bins=30)
plt.tight_layout()
plt.show()

print(df.columns)

# Scatter plot between Tot_CredExp and VintDel
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Tot_CredExp', y='VintDel', data=df)
plt.title('Scatter Plot between Tot_CredExp and VintDel')
plt.show()

# Scatter plot between Tot_CredExp and Del_Rate
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Tot_CredExp', y='Del_Rate', data=df)
plt.title('Scatter Plot between Tot_CredExp and Del_Rate')
plt.show()
```

```
# Box plot for Tot_CredExp
plt.figure(figsize=(10, 6))
sns.boxplot(x='Tot_CredExp', data=df)
plt.title('Box Plot of Tot_CredExp')
plt.show()

# Box plot for Del_Rate
plt.figure(figsize=(10, 6))
sns.boxplot(x='Del_Rate', data=df)
plt.title('Box Plot of Del_Rate')
plt.show()

import matplotlib.pyplot as plt
import pandas as pd

# Check if 'Year' column exists
if 'Year' in df.columns:
    # Ensure 'Year' is in datetime format
    df['Year'] = pd.to_datetime(df['Year'])

    # Plot 'Tot_CredExp' over time
    plt.figure(figsize=(12, 6))
    df.set_index('Year')['Tot_CredExp'].plot()
    plt.title('Time Series Plot of Tot_CredExp')
    plt.xlabel('Date')
    plt.ylabel('Tot_CredExp')
    plt.show()

    # Plot 'VintDel' over time
    plt.figure(figsize=(12, 6))
    df.set_index('Year')['VintDel'].plot()
    plt.title('Time Series Plot of VintDel')
    plt.xlabel('Date')
    plt.ylabel('VintDel')
    plt.show()
else:
    print("Column 'Date' not found in the DataFrame.")
```

- **Histogram Plotting:**
  - **Objective:** Visualize the distribution of each feature in the DataFrame.
  - **Method:** Uses `hist` from `pandas` to create histograms for all columns, with 30 bins and a plot size of 12×10 inches.
- **Scatter Plots:**
  - **Scatter Plot 1:** `Tot_CredExp` vs. `VintDel` to examine their relationship.
  - **Scatter Plot 2:** `Tot_CredExp` vs. `Del_Rate` to visualize the correlation between these variables.
  - **Method:** Uses `seaborn.scatterplot` for plotting, with titles for clarity.
- **Box Plots:**
  - **Box Plot 1:** Distribution of `Tot_CredExp`.
  - **Box Plot 2:** Distribution of `Del_Rate`.
  - **Purpose:** To identify outliers and understand the spread of these variables.
  - **Method:** Uses `seaborn.boxplot`.
- **Time Series Plots (Conditional on 'Year' Column):**
  - **Objective:** Visualize the trend of `Tot_CredExp` and `VintDel` over time if the `Year` column is present.
  - **Method:**
    - Converts the `Year` column to datetime format.
    - Plots `Tot_CredExp` and `VintDel` against time using the `plot` method.
    - If the `Year` column does not exist, an error message is printed.

```
# Calculate the correlation matrix
corr_matrix = df.corr()

# Correlation of features with target variable
target_corr = corr_matrix['Tot_CredExp'].sort_values(ascend
ing=False)
print("Correlation with Tot_CredExp:\n", target_corr)
```

```
# Correlation of features with target variable
target_corr = corr_matrix['VintDel'].sort_values(ascending=
False)
print("Correlation with VintDel:\n", target_corr)

# Correlation of features with target variable
target_corr = corr_matrix['Growth_Rate'].sort_values(ascend
ing=False)
print("Correlation with Growth_Rate:\n", target_corr)

# Descriptive statistics
print(df.describe())
```

- **Correlation Matrix Calculation:**
  - **Objective:** Compute the correlation matrix for all numerical features in the DataFrame.
  - **Method:** Uses `df.corr()` to get the correlation values between features.
- **Correlation with Specific Variables:**
  - **Correlation with `Tot_CredExp`:** Displays correlations of all features with `Tot_CredExp`, sorted in descending order.
  - **Correlation with `VintDel`:** Displays correlations of all features with `VintDel`, sorted in descending order.
  - **Correlation with `Growth_Rate`:** Displays correlations of all features with `Growth_Rate`, sorted in descending order.
- **Descriptive Statistics:**
  - **Objective:** Obtain summary statistics for each numerical feature.
  - **Method:** Uses `df.describe()` to get counts, means, standard deviations, min/max values, and quartiles.

9. Scaling and Prediction

```
from scipy import stats

# T-test between two groups
group1 = df[df['Tot_CredExp'] == 'A']['VintDel']

group2 = df[df['Tot_CredExp'] == 'B']['VintDel']

t_stat, p_val = stats.ttest_ind(group1, group2)
print(f"T-statistic: {t_stat}, P-value: {p_val}")
```

```
from scipy.stats import chi2_contingency
from sklearn.preprocessing import StandardScaler

# Scale the target variable
scaler_y = StandardScaler()
y_train_scaled = scaler_y.fit_transform(y_train.values.resh
ape(-1, 1))

# Now, when fitting the model, ensure you're using the Data
Frame format with feature names
model = RandomForestRegressor(random_state=42)
model.fit(X_train_scaled, y_train_scaled.ravel())  # Flatte
n y_train_scaled

# Predict on test data
y_pred_scaled = model.predict(X_test_scaled)
y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-
1, 1))
y_test_original = scaler_y.inverse_transform(y_test.values.
reshape(-1, 1))

# Print predictions for checking
print("Predicted values (original scale):", y_pred.flatten
())
print("True values (original scale):", y_test_original.flat
ten())
```

```
# Chi-Square test
contingency_table = pd.crosstab(df['Tot_CredExp'], df['Vint
Del'])
chi2_stat, p_val, dof, ex = chi2_contingency(contingency_ta
ble)
print(f"Chi-Square Statistic: {chi2_stat}, P-value: {p_va
l}")
```

```
from scipy import stats
# Perform ANOVA
anova_result = stats.f_oneway(
    df[df['Year'] == 'A']['Tot_CredExp'],
    df[df['Year'] == 'B']['Tot_CredExp'],
    df[df['Year'] == 'C']['Tot_CredExp']
)

# Extracting the results
f_statistic = anova_result.statistic
p_value = anova_result.pvalue

print(f"ANOVA F-Statistic: {f_statistic}")
print(f"ANOVA P-Value: {p_value}")

# Interpretation
if p_value < 0.05:
    print("There is a significant difference between the me
ans of the groups.")
else:
    print("There is no significant difference between the m
eans of the groups.")
```

- **Scaling the Target Variable:**
  - **Objective:** Standardize the target variable (`y_train`) for better model performance.
  - **Method:**

        - **Standardization:** Uses `StandardScaler` to scale `y_train`.
        - **Reshaping:** Converts `y_train` to a 2D array before scaling and flattens it back after scaling.
- **Model Training and Prediction:**
    - **Model:** Trains a `RandomForestRegressor` using the scaled training features (`X_train_scaled`) and scaled target (`y_train_scaled`).
    - **Prediction:**
        - Predicts on the scaled test set (`X_test_scaled`).
        - Converts the predictions back to the original scale using `scaler_y.inverse_transform`.
        - Compares and prints the predicted and true values in the original scale.
- **Chi-Square Test:**
    - **Objective:** Assess the association between `Tot_CredExp` and `VintDel` using a contingency table.
    - **Method:**
        - **Contingency Table:** Creates a cross-tabulation of `Tot_CredExp` and `VintDel`.
        - **Chi-Square Test:** Uses `chi2_contingency` to calculate the chi-square statistic, p-value, degrees of freedom, and expected frequencies.
        - **Output:** Prints the chi-square statistic and p-value to evaluate if there is a significant association between the two variables.
- **ANOVA Test:**
    - **Objective:** Compare the means of `Tot_CredExp` across three different years ('A', 'B', 'C') to determine if there are significant differences among them.
    - **Method:**
        - **ANOVA Test:** Uses `stats.f_oneway` to perform a one-way ANOVA test across the three groups.
        - **Extract Results:**

        - **F-Statistic:** Measures the ratio of variance between groups to variance within groups.
        - **P-Value:** Assesses the probability that the observed differences are due to chance.
- **Results and Interpretation:**
    - **Output:** Prints the ANOVA F-Statistic and p-value.
    - **Significance Check:** Compares the p-value with a significance level of 0.05:
        - If `p-value < 0.05`: Indicates a significant difference in means between at least two groups.
        - If `p-value >= 0.05`: Indicates no significant difference in means between the groups.

10. Regularization using Neural Network

```
import tensorflow as tf
from tensorflow.keras import layers, regularizers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import KFold
import numpy as np

# Load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_test = X_train / 255.0, X_test / 255.0

# Flatten the images
X_train = X_train.reshape(-1, 28 * 28)
X_test = X_test.reshape(-1, 28 * 28)

# Define the model creation function with regularization
def create_model():
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(7
```

```
84,),
                    kernel_regularizer=regularizers.l2(0.0
01)),
        layers.Dropout(0.5),
        layers.BatchNormalization(),
        layers.Dense(256, activation='relu',
                    kernel_regularizer=regularizers.l2(0.0
01)),
        layers.Dropout(0.5),
        layers.BatchNormalization(),
        layers.Dense(128, activation='relu',
                    kernel_regularizer=regularizers.l2(0.0
01)),
        layers.Dropout(0.5),
        layers.BatchNormalization(),
        layers.Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Set up K-Fold Cross-Validation
kfold = KFold(n_splits=5, shuffle=True)
fold_no = 1
cv_scores = []

for train, val in kfold.split(X_train, y_train):
    model = create_model()

    # Early stopping
    early_stopping = EarlyStopping(monitor='val_loss', pati
ence=3, restore_best_weights=True)

    # Fit model
    history = model.fit(X_train[train], y_train[train],
                        epochs=50,
```

```
                          batch_size=64,
                          validation_data=(X_train[val], y_tr
ain[val]),
                          callbacks=[early_stopping],
                          verbose=1)

    # Evaluate model
    scores = model.evaluate(X_train[val], y_train[val], ver
bose=0)
    print(f"Score for fold {fold_no}: {model.metrics_names
[1]} of {scores[1]*100}%")
    cv_scores.append(scores[1] * 100)
    fold_no += 1

# Calculate and print average cross-validation score
print(f"Average cross-validation accuracy: {np.mean(cv_scor
es)}%")

# Final evaluation on test data
final_model = create_model()
final_model.fit(X_train, y_train, epochs=50, batch_size=64,
verbose=1)
test_loss, test_acc = final_model.evaluate(X_test, y_test,
verbose=0)
print(f"Test accuracy: {test_acc * 100}%")
```

- **Dataset Preparation:**
  - **Data Loading:** Uses the MNIST dataset from `tensorflow.keras.datasets`.
  - **Normalization:** Scales pixel values to the range [0, 1].
  - **Flattening:** Reshapes images from 28×28 to a 1D array of 784 pixels.
- **Model Definition (`create_model` function):**
  - **Architecture:**
    - **Input Layer:** Dense layer with 512 units, ReLU activation, L2 regularization.
    - **Hidden Layers:**

        - Dense layer with 256 units, ReLU activation, L2 regularization.
        - Dense layer with 128 units, ReLU activation, L2 regularization.
      - **Dropout Layers:** Applied after each dense layer with a rate of 0.5 to prevent overfitting.
      - **Batch Normalization:** Applied after each dropout layer to normalize activations.
      - **Output Layer:** Dense layer with 10 units (for 10 classes), Softmax activation.
    - **Compilation:** Uses Adam optimizer, sparse categorical crossentropy loss, and accuracy metric.
- **K-Fold Cross-Validation:**
    - **Setup:** Splits data into 5 folds.
    - **Training and Validation:**
      - **Early Stopping:** Monitors validation loss and stops training if it doesn't improve for 3 epochs, restoring the best weights.
      - **Training:** Trains model for up to 50 epochs with batch size 64 on training data and validates on validation data.
      - **Evaluation:** Evaluates model performance on validation data for each fold, recording accuracy.
    - **Output:** Prints accuracy for each fold and the average cross-validation accuracy.
- **Final Model Training and Evaluation:**
    - **Training:** Trains a new model on the entire training dataset.
    - **Testing:** Evaluates final model on the test dataset and prints test accuracy.

This code block implements a neural network for image classification using the MNIST dataset, applies cross-validation to evaluate model performance, and performs final training and testing on the dataset.

```
plt.figure(figsize=(12, 6))
```

```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

Plot Training & Validation Loss:

- **Training Loss:** Plots the loss values recorded during training from the `history` object.
- **Validation Loss:** Plots the loss values recorded during validation from the `history` object.
- **Title:** Labels the plot as 'Model Loss'.
- **Axes Labels:** Sets the x-axis label to 'Epoch' and the y-axis label to 'Loss'.
- **Legend:** Adds a legend to differentiate between training and validation loss curves.
- **Display:** Shows the plot.

11. Classifier accuracy scores

```
from sklearn.ensemble import BaggingClassifier, GradientBoo
stingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

# Load example dataset
data = load_iris()
X = data.data
y = data.target
```

```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, t
est_size=0.3, random_state=42)

# Initialize the base model
base_model = DecisionTreeClassifier()

# Initialize the bagging model
bagging_model = BaggingClassifier(base_estimator=base_mode
l, n_estimators=50, random_state=42)

# Train the bagging model
bagging_model.fit(X_train, y_train)

# Predict and evaluate the Bagging model
y_pred_bagging = bagging_model.predict(X_test)
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
print(f"Accuracy of Bagging Classifier: {accuracy_bagging:.
2f}")

# Initialize and train the Gradient Boosting model
boosting_model = GradientBoostingClassifier(n_estimators=10
0, learning_rate=0.1, random_state=42)
boosting_model.fit(X_train, y_train)

# Predict and evaluate the Boosting model
y_pred_boosting = boosting_model.predict(X_test)
accuracy_boosting = accuracy_score(y_test, y_pred_boosting)
print(f'Boosting Model Accuracy: {accuracy_boosting:.2f}')
```

- **Bagging Classifier:**
  - **Base Model:** A `DecisionTreeClassifier` is used as the base estimator.
  - **Bagging Model:** `BaggingClassifier` with 50 estimators is initialized and trained.
  - **Evaluation:** Calculates and prints the accuracy of the Bagging model on the test set.
- **Gradient Boosting Classifier:**

    - **Model:** `GradientBoostingClassifier` with 100 estimators and a learning rate of 0.1 is initialized and trained.
    - **Evaluation:** Calculates and prints the accuracy of the Gradient Boosting model on the test set.
- **Bagging:** Aggregates predictions from multiple models (here, decision trees) to improve overall performance and reduce variance.
- **Boosting:** Sequentially builds models where each new model attempts to correct the errors made by the previous ones, often leading to improved performance.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define the model
model = GradientBoostingClassifier()

# Define the parameter distribution
param_dist = {
    'n_estimators': randint(50, 200),
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': randint(3, 10)
}

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=model, param_d
istributions=param_dist, n_iter=50, cv=3, n_jobs=-1, verbos
e=2, random_state=42)

# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)

# Best parameters and score
print(f'Best Parameters: {random_search.best_params_}')
print(f'Best Score: {random_search.best_score_:.2f}')
```

- **Model Definition:**
  - `GradientBoostingClassifier()` is defined as the model to be tuned.
- **Parameter Distribution:**
  - `param_dist` specifies the ranges of hyperparameters to be explored:
    - `n_estimators`: Number of boosting stages to be used (between 50 and 200).
    - `learning_rate`: Step size at each iteration (0.01, 0.1, 0.2).
    - `max_depth`: Maximum depth of the individual trees (between 3 and 10).
- **RandomizedSearchCV:**
  - `RandomizedSearchCV` is initialized with:
    - `estimator`: The `GradientBoostingClassifier` model.
    - `param_distributions`: The hyperparameter ranges.
    - `n_iter`: Number of parameter settings to sample (50).
    - `cv`: Number of cross-validation folds (3).
    - `n_jobs`: Number of jobs to run in parallel (-1 means using all available processors).
    - `verbose`: Level of verbosity (2 for detailed logs).
    - `random_state`: Seed for reproducibility (42).
- **Fitting the Model:**
  - `random_search.fit(X_train, y_train)` performs the search over the specified hyperparameters and fits the model on the training data.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, Strat
ifiedKFold, GridSearchCV, RandomizedSearchCV, cross_val_sco
re
from sklearn.preprocessing import StandardScaler, LabelEnco
der
```

```
from sklearn.ensemble import RandomForestClassifier, Gradie
ntBoostingClassifier, VotingClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import RandomOverSampler
import xgboost as xgb

# Load data
df = pd.read_csv("D:/RI IIM INDR Files/Market report Cred E
xp_filled.csv")

# Preprocess data
df = df.drop(columns=['Month'])
X = df.drop(columns=['CredExp_Class'])
y = df['CredExp_Class']

# Handle classes with very few samples by merging them
min_class_count = 3
class_counts = y.value_counts()
classes_to_merge = class_counts[class_counts < min_class_co
unt].index
if len(classes_to_merge) > 0:
    y = y.replace(classes_to_merge, 'Other')

# Impute, scale, and encode features
X_imputed = SimpleImputer(strategy='mean').fit_transform(X)
X_scaled = StandardScaler().fit_transform(X_imputed)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X_scale
d, y_encoded, test_size=0.3, random_state=42)

# Apply RandomOverSampler for class balancing
ros = RandomOverSampler(random_state=42)
X_train_balanced, y_train_balanced = ros.fit_resample(X_tra
```

```
in, y_train)

# Define cross-validation strategy
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state
=42)

# Define models
rf = RandomForestClassifier(class_weight='balanced')
gb = GradientBoostingClassifier()
svc = SVC(kernel='rbf', probability=True, class_weight='bal
anced')
xgb_model = xgb.XGBClassifier()

# Create Voting Classifier
voting_clf = VotingClassifier(estimators=[
    ('rf', rf),
    ('gb', gb),
    ('svc', svc)
], voting='soft')

# Hyperparameter tuning for Random Forest
param_grid_rf = {'n_estimators': [50, 100, 200], 'max_dept
h': [None, 10, 20]}
grid_search_rf = GridSearchCV(estimator=rf, param_grid=para
m_grid_rf, cv=cv, scoring='accuracy')
grid_search_rf.fit(X_train_balanced, y_train_balanced)
best_rf = grid_search_rf.best_estimator_

# Hyperparameter tuning for Gradient Boosting
param_distributions_gb = {'n_estimators': [50, 100, 200],
'max_depth': [None, 10, 20]}
random_search_gb = RandomizedSearchCV(estimator=gb, param_d
istributions=param_distributions_gb, n_iter=9, cv=cv, scori
ng='accuracy')
random_search_gb.fit(X_train_balanced, y_train_balanced)
best_gb = random_search_gb.best_estimator_

# Train models
```

```
best_rf.fit(X_train_balanced, y_train_balanced)
best_gb.fit(X_train_balanced, y_train_balanced)
voting_clf.fit(X_train_balanced, y_train_balanced)
xgb_model.fit(X_train_balanced, y_train_balanced)

# Make predictions
y_pred_rf = best_rf.predict(X_test)
y_pred_gb = best_gb.predict(X_test)
y_pred_voting = voting_clf.predict(X_test)
y_pred_xgb = xgb_model.predict(X_test)

# Decode the predictions back to original labels
y_test_decoded = label_encoder.inverse_transform(y_test)
y_pred_rf_decoded = label_encoder.inverse_transform(y_pred_
rf)
y_pred_gb_decoded = label_encoder.inverse_transform(y_pred_
gb)
y_pred_voting_decoded = label_encoder.inverse_transform(y_p
red_voting)
y_pred_xgb_decoded = label_encoder.inverse_transform(y_pred
_xgb)

# Convert predictions and test labels to strings
y_test_decoded = [str(label) for label in y_test_decoded]
y_pred_rf_decoded = [str(label) for label in y_pred_rf_deco
ded]
y_pred_gb_decoded = [str(label) for label in y_pred_gb_deco
ded]
y_pred_voting_decoded = [str(label) for label in y_pred_vot
ing_decoded]
y_pred_xgb_decoded = [str(label) for label in y_pred_xgb_de
coded]

# Check unique values
print(f"Unique values in y_test_decoded: {np.unique(y_test_
decoded)}")
print(f"Unique values in y_pred_rf_decoded: {np.unique(y_pr
ed_rf_decoded)}")
```

```
# Evaluate models
print("Random Forest Classification Report:")
print(classification_report(y_test_decoded, y_pred_rf_decod
ed, zero_division=1))

print("Gradient Boosting Classification Report:")
print(classification_report(y_test_decoded, y_pred_gb_decod
ed, zero_division=1))

print("Voting Classifier Classification Report:")
print(classification_report(y_test_decoded, y_pred_voting_d
ecoded, zero_division=1))

print("XGBoost Classification Report:")
print(classification_report(y_test_decoded, y_pred_xgb_deco
ded, zero_division=1))

# Cross-validation scores using StratifiedKFold
cv_scores_rf = cross_val_score(best_rf, X_scaled, y_encode
d, cv=cv, scoring='accuracy')
print("Random Forest Cross-Val Accuracy: {:.2f}".format(cv_
scores_rf.mean()))

cv_scores_gb = cross_val_score(best_gb, X_scaled, y_encode
d, cv=cv, scoring='accuracy')
print("Gradient Boosting Cross-Val Accuracy: {:.2f}".format
(cv_scores_gb.mean()))

cv_scores_voting = cross_val_score(voting_clf, X_scaled, y_
encoded, cv=cv, scoring='accuracy')
print("Voting Classifier Cross-Val Accuracy: {:.2f}".format
(cv_scores_voting.mean()))

cv_scores_xgb = cross_val_score(xgb_model, X_scaled, y_enco
ded, cv=cv, scoring='accuracy')
print("XGBoost Cross-Val Accuracy: {:.2f}".format(cv_scores
_xgb.mean()))
```

- **Model Definition and Hyperparameter Tuning:**
  - **Models:** Defines several classifiers:
    - `RandomForestClassifier`
    - `GradientBoostingClassifier`
    - `SVC` (Support Vector Classifier)
    - `XGBoost` (Extreme Gradient Boosting)
  - **Voting Classifier:** Combines predictions from `RandomForestClassifier`, `GradientBoostingClassifier`, and `SVC` using soft voting.
  - **Hyperparameter Tuning:**
    - `GridSearchCV` for `RandomForestClassifier`.
    - `RandomizedSearchCV` for `GradientBoostingClassifier`.
- **Training Models:**
  - Trains each model on the balanced training data.
- **Making Predictions and Evaluating Models:**
  - **Predictions:** Uses each trained model to predict the test data.
  - **Decoding:** Converts encoded predictions and test labels back to original labels.
  - **Classification Reports:** Evaluates and prints classification reports for each model.
  - **Cross-Validation Scores:** Computes cross-validation accuracy for each model using `StratifiedKFold`.

12. Prediction plots

```
import matplotlib.pyplot as plt

# Create example time-series data
dates = np.arange('2023-01', '2024-01', dtype='datetime64
[M]')
historical_data = np.sin(np.linspace(0, 10, len(dates)))  #
```

```
Example historical data
predicted_trends = np.sin(np.linspace(0, 10.5, len(dates)))
# Example future trends

plt.figure(figsize=(12, 6))
plt.plot(dates, historical_data, label='Historical Data', c
olor='blue')
plt.plot(dates, predicted_trends, label='Predicted Trends',
color='red', linestyle='--')
plt.xlabel('Date')
plt.ylabel('Value')
plt.title('Historical Data vs Predicted Trends')
plt.legend()
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```

- **Purpose:** To visualize how historical data compares with predicted trends over time.
- **Data Generation:**
  - `dates` creates a range of monthly dates from January 2023 to January 2024.
  - `historical_data` and `predicted_trends` are generated as sine waves to simulate time-series data.
- **Plotting:**
  - `plt.plot()` is used to plot the historical data and predicted trends.
  - `label`, `color`, `linestyle`, etc., customize the plot.
  - `plt.xticks(rotation=45)` rotates date labels for better readability.

```
from sklearn.model_selection import learning_curve
import matplotlib.pyplot as plt

train_sizes, train_scores, test_scores = learning_curve(mod
el, X_train, y_train, cv=5, n_jobs=-1)
```

```
plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', colo
r='red', label='Training score')
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', color
='green', label='Cross-validation score')
plt.xlabel('Training examples')
plt.ylabel('Score')
plt.title('Learning Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

- **Purpose:** To visualize how the training and cross-validation scores change with varying training set sizes.
- **Functionality:**
  - `learning_curve()` computes training and validation scores for different training sizes.
  - `train_sizes` represents the number of training samples.
  - `train_scores` and `test_scores` are arrays containing training and cross-validation scores, respectively.
- **Plotting:**
  - `plt.plot()` plots both training scores (red) and cross-validation scores (green) against training sizes.
  - `plt.xlabel()` and `plt.ylabel()` label the axes.
  - `plt.legend()` adds a legend to the plot for clarity.