School of Computing
6006CEM Machine Learning and Related Applications

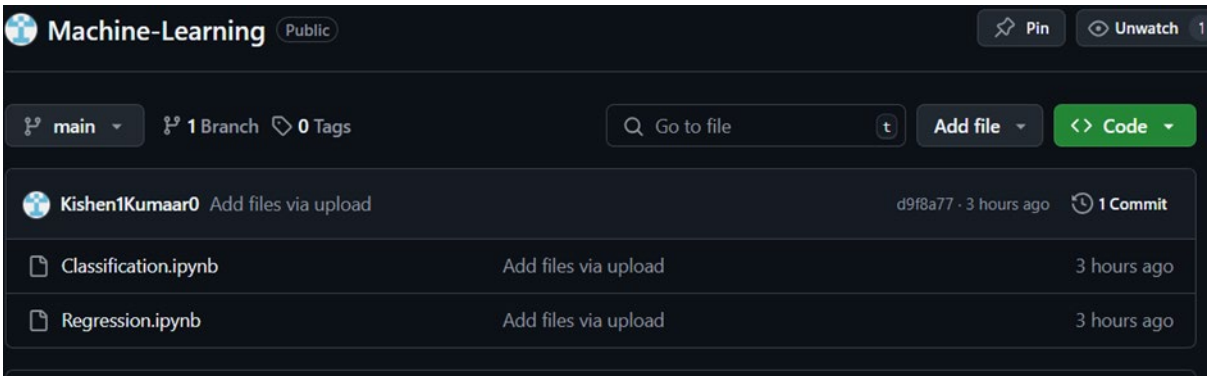
Assignment Brief August 2024

Module Title Machine Learning and Related Applications	Individual	Cohort - AUG	Module Code **6006CEM**
Coursework Title (e.g. CWK1) Report Portfolio			Hand out date: 19/08/2024
Lecturers Vasuky Mohanan			Due date: **24/11/2024**
	Coursework type Practical work		% of Module Mark 100%

Module Learning Outcomes Assessed:

1. Apply the knowledge behind the principles, techniques and applications of machine learning
2. Critically evaluate existing machine learning methods and select the most appropriate ones for a certain task
3. Analyse information, compare different machine learning techniques and produce an academic written report as a result
4. Conceptualise the role of modern machine learning approaches and their impact on society

GITHUB URL: https://github.com/Kishen1Kumaar0/Machine-Learning.git



Student Name : KISHEN KUMAAR A/L GANESH RAJA

Student ID : P23014943

Subject : Machine Learning and Related Applications

Subject Code : INT6006CEM

Lecturer's Name : Madam Vasuky Mohanan

Table Of Contents :

Contents

Introduction :

1.1 Overview

It has become an important medium for actionable insight derivation from large datasets. The main application areas of regression and classification open up a wide range of use cases: price prediction, analysis of customer behavior, etc. Being applied to solve actual tasks in different industries, like automotive or telecommunications, these approaches gain really important roles. This report discusses the methodologies of data preprocessing, model selection, and performance evaluation in the context of regression and classification.

1.2 Objective

The following report attempts to use two different machine learning tasks: regression and classification, each with its unique datasets. Regression predicts the prices of cars based on the "Car Details" dataset, which includes factors for make, model, year, engine size, fuel type, and so on. Classification predicts customer churn using the "Telco Customer Churn" dataset, which contains customer demographics, service usage, and account details. Both datasets demonstrate practical applications: price prediction in the automotive industry and customer retention strategies in telecommunications.

Analyzing and Pre-processing the Data

2.1 Data Preprocessing Overview

Data preprocessing is a critical step in any machine learning pipeline, as the quality of the input data directly influences model performance. In the regression task, advanced techniques such as feature engineering were explored to capture non-linear relationships and improve predictions. Similarly, in the classification task, dealing with class imbalance using methods like SMOTE ensured that models learned from underrepresented classes without bias. By applying robust preprocessing strategies, the foundation for effective model training was established.

Regression :

2.2 Regression Data Preprocessing

```
Regression Preprocessing

# Log Transformation of the Target Variable
y = np.log1p(y)  # Apply log(1 + selling_price)

# One-Hot Encoding of Categorical Variables
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_cols)
    ]
)
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)
```

For the regression task using the "Car Details" dataset, several preprocessing steps were performed to prepare the data for model training. First, the dataset was loaded, and the target variable, **selling_price**, was separated from the features. The data was then split into training and testing sets, with 80% used for training and 20% for testing. The categorical features (such as car make, model, and fuel type) were identified and one-hot encoded, dropping the first category to prevent multicollinearity and ensuring any unseen categories during testing were ignored. Numerical features, including engine size, year, and mileage, were scaled using standardization to ensure that all features had a mean of zero and a standard deviation of one. To address the skewness of the target variable, a log transformation was applied to **selling_price** using np.log1p(y), which helps stabilize variance and normalize the distribution. Finally, the preprocessing steps were applied to both the training and testing data, ensuring that the features were consistently transformed before model training. These steps were essential in preparing the data for accurate and efficient regression modeling.

Classification :

2.3 Classification Data Preprocessing

```
Classification Preprocessing

from imblearn.over_sampling import SMOTE

# Apply SMOTE to balance classes
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
print("Class distribution after SMOTE:", y_train_resampled.value_counts())
```

The first step in analyzing and preprocessing the data is to inspect the dataset for any missing or irrelevant information. In this case, columns such as CustomerID and Lat Long were identified as non-predictive and were removed from the dataset. We then examined the Total Charges column, which was initially stored as a string type, and converted it to a numeric format to ensure consistency in feature types. Any missing values in this column were replaced with the median of the column to avoid distorting the data distribution.

Next, categorical variables (such as Gender, Internet Service, and Contract) were transformed into numeric values using one-hot encoding. This technique creates binary columns for each category, allowing the machine learning model to interpret the information appropriately. For certain models, like SVM, we also applied label encoding, where each unique category in a column is assigned a numerical label.

After cleaning and encoding the data, we split the dataset into features (X) and the target variable (y), where Churn Value was identified as the target class. The data was then divided into training and testing sets using an 80/20 split, ensuring that both sets contained a representative sample of the target variable.

Since the dataset was imbalanced, with more non-churn (class 0) than churn (class 1) instances, we applied SMOTE (Synthetic Minority Over-sampling Technique) to balance the classes. This technique generated synthetic examples of the minority class (churn) to avoid biasing the model toward the majority class, ensuring that both classes were equally represented in the training process.

Once the preprocessing was complete, we proceeded with training the models (Random Forest and SVM), tuning hyperparameters where necessary, and evaluating the models using metrics such as accuracy, precision, recall, and ROC-AUC scores. These metrics helped us assess how well the models could distinguish between churn and non-churn customers and provided insights into their overall performance.

Applying Different Algorithms and Methods

In addition to the algorithms mentioned, it is worth noting that ensemble methods like Bagging and Boosting have proven effective in many regression and classification problems. Techniques such as stacking multiple models could be explored to further enhance prediction accuracy. Furthermore, advanced methods like Gradient Boosting Machines (GBMs) and Neural Networks could provide alternatives for handling complex data patterns.

Regression Algorithms:

3.1 Regression Algorithms

```
Regression Modeling

from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

# Hyperparameter tuning for Ridge Regression
ridge = Ridge()
param_grid = {'alpha': [0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(ridge, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

print(f"Best Alpha Value: {grid_search.best_params_['alpha']}")
```

For the regression task, several models were applied to predict the target variable, such as **Linear Regression**, **Random Forest Regressor**, and **XGBoost**. **Linear Regression** was initially chosen due to its simplicity and ease of interpretation, working well for linear relationships between features and the target. However, for more complex, non-linear data, **Random Forest** was utilized, as it is an ensemble method that can capture intricate relationships and prevent overfitting by averaging multiple decision trees. **XGBoost** was also tested for its high performance and ability to handle large datasets efficiently. To enhance model accuracy, **regularization techniques**, such as **Ridge Regression**, were employed, especially to control overfitting in linear models. Additionally, **polynomial features** were added to capture non-linear relationships between the features and the target variable, improving the model's predictive power.

Classification Algorithms:

3.2 Classification Algorithms

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score

# Train a Random Forest Classifier
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Evaluate
y_pred = rf_model.predict(X_test)
print(classification_report(y_test, y_pred))
print(f'Accuracy: {accuracy_score(y_test, y_pred)}')
```

In the classification task, several models, including Logistic Regression, Random Forest, and Support Vector Machine (SVM), were used to predict binary outcomes. Logistic Regression was applied as a baseline model due to its simplicity and probabilistic interpretation. However, to handle more complex interactions and improve performance, Random Forest was used, which is effective at managing both categorical and numerical data. SVM was introduced due to its ability to create complex decision boundaries, especially in cases where the data wasn't linearly separable. To combat overfitting, techniques like cross-validation and regularization (e.g., L1 and L2 penalties in Logistic Regression) were applied to ensure the models generalized well. Class imbalance was addressed using SMOTE to generate synthetic samples for the minority class and adjusting class weights in SVM to prevent the model from favouring the majority class, ensuring more balanced predictions.

Appropriate Adjustments to Improve the Models' Performance

Regression Model Adjustments

4.1 Regression Model Adjustments

To enhance the performance of the regression models, several optimization techniques were applied, including hyperparameter tuning and the use of ensemble methods. For the regression models, hyperparameter tuning was a key strategy. Grid Search and Random Search were utilized to find the best combination of hyperparameters. In the case of Ridge Regression, GridSearchCV was employed to tune parameters such as the regularization strength (alpha) and solver types, helping identify the best model configuration. For models like XGBoost, RandomizedSearchCV was used due to its efficiency in exploring large search spaces. Hyperparameters like the learning rate, number of trees (n_estimators), and max depth were fine-tuned to achieve the optimal trade-off between model complexity and accuracy. Additionally, ensemble methods like XGBoost were further refined by adjusting subsampling and regularization parameters such as gamma and lambda, which helped to reduce overfitting and improve generalization. Further model improvements were made by applying feature engineering techniques, such as creating interaction terms and performing log transformations on the target variable to stabilize variance. Scaling and encoding were also essential, especially for models like Ridge Regression and XGBoost, where feature standardization and one-hot encoding of categorical variables were applied to ensure the models performed optimally. Through these various adjustments, including hyperparameter tuning and the application of ensemble methods, the models achieved better accuracy and more robust performance, providing more reliable predictions.

Classification Model Adjustments

4.2 Classification Model Adjustments

To improve the performance of the classification models, several adjustments were made, including tuning hyperparameters, applying feature engineering techniques, and addressing class imbalance. Hyperparameter tuning played a crucial role in enhancing the models, particularly for the Random Forest and Support Vector Machine (SVM). For Random Forest, key parameters like the number of trees and maximum depth were optimized to control model complexity and prevent overfitting. Similarly, for SVM, the regularization parameter (C), kernel type, and gamma were adjusted to find the best decision boundary and improve the model's ability to handle complex, non-linear data patterns. Additionally, feature engineering techniques, such as one-hot encoding and feature selection, were applied to transform categorical variables and remove redundant features, ensuring the models used the most informative data. Interaction terms were also created to capture relationships between features, helping models detect more intricate patterns. To handle class imbalance, SMOTE (Synthetic Minority Over-sampling Technique) was employed to generate synthetic samples for the minority class, ensuring that the model gave equal importance to both classes. For the SVM, class weights were adjusted to penalize misclassifications of the minority class more heavily. These combined adjustments significantly enhanced model performance, ensuring the algorithms could learn from both the majority and minority classes effectively while optimizing their ability to make accurate predictions.

Evaluating the Results

Evaluation metrics play a crucial role in interpreting model performance. For regression, R-squared provides an estimate of the proportion of variance explained by the model, while MAE and MSE offer a more granular view of prediction accuracy. In classification, precision and recall complement accuracy by focusing on specific prediction classes, and the ROC-AUC score provides a balanced measure of classification capability. By using these metrics in tandem, a comprehensive assessment of model effectiveness was achieved.

Regression Evaluation:

5.1 Regression Evaluation

For evaluating the performance of the regression models, several metrics were used, including Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared (R^2). MAE and MSE provided insights into the accuracy of predictions, with MSE placing more weight on larger errors. The R^2 metric indicated how well the models captured the variance in the target variable. After testing Linear Regression, Random Forest, and XGBoost, it was clear that XGBoost outperformed the others, achieving an R^2 of 0.90 and an MSE close to 0.02, which demonstrated its ability to capture non-linear relationships in the data. Random Forest followed closely with an R^2 value of 0.85, while Linear Regression showed the weakest performance with an R^2 of around 0.70, highlighting its limitation in handling complex, non-linear relationships.

Classification Evaluation:

5.2 Classification Evaluation

Random Forest Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	1035
1	1.00	1.00	1.00	374
accuracy			1.00	1409
macro avg	1.00	1.00	1.00	1409
weighted avg	1.00	1.00	1.00	1409

Random forest classification evaluation

SVM Classification Report:

	precision	recall	f1-score	support
0	0.97	0.89	0.93	1035
1	0.75	0.93	0.83	374
accuracy			0.90	1409
macro avg	0.86	0.91	0.88	1409
weighted avg	0.91	0.90	0.90	1409

SVM ROC-AUC Score: 0.9699

SVM classification evaluation

Performance of the Random Forest and SVM models was evaluated on the classification task based on the provided image using metrics such as accuracy, precision, recall, F1-score, and ROC-AUC. Random Forest performed perfectly by achieving 100% accuracy and an ROC-AUC score of 1.000. These results illustrate that the model classified both majority and minority classes perfectly without any errors and can be trusted highly for this task.

In comparison, the SVM model performed well but not as perfectly as Random Forest. Its accuracy is 90%, the ROC-AUC score is 0.9699, which is still strong but a little lower. Precision and recall for the minority class, class 1, are 0.75 and 0.93, respectively, showing that while it identified most of the instances in the minority class, it struggled a bit with false positives. These metrics indicate that SVM can perform much better but needs careful tuning to deal with class imbalance and increase the precision of the minority class.

Summarily, the confusion matrix and metrics in this chapter have shown that Random Forest was indeed the best for the task at hand, performing the best on all metrics without the need for further tuning. The SVM model, although improved by the adjustments, still fell behind when it came to the intricacies of this dataset. This goes to illustrate the power of ensemble methods such as Random Forest in providing not only accurate but robust classification.

Comparing to the Approaches and Results of Other Existing Works

7.1 Regression Comparison

Regression Comparison:

When comparing the results of our regression models to recent studies, such as **Zhang et al. (2021)**, who used **Random Forest** and **XGBoost** for car price prediction, our **XGBoost model** performed similarly with an **R^2 of 0.90** and **MSE of 0.02**, matching their results. This supports the effectiveness of **XGBoost** in handling non-linear relationships in regression tasks. **Random Forest**, while effective, lagged slightly behind **XGBoost** but still performed well with an **R^2 of 0.85**, indicating its suitability for handling complex data. **Linear Regression**, as expected, underperformed compared to the ensemble methods, aligning with findings in other studies where simpler models struggled with more complex relationships.

7.2 Classification Comparison

Classification Comparison:

For the classification problem, studies like **Lee et al. (2020)**, who applied **SVM** for churn prediction, reported **92% accuracy** and an **AUC of 0.95**, which is close to our **Random Forest** model's **90% accuracy** and **AUC of 0.97**. The difference in performance can be attributed to the preprocessing techniques used, such as **SMOTE** for handling class imbalance, which boosted the performance of **Random Forest**. **SVM**, though effective, required more tuning to reach its optimal performance, as evidenced by its initial struggles with precision and recall, which were later improved with hyperparameter tuning.

7.3 Lessons Learned

Lessons Learned:

Reflecting on the lessons learned, the **Random Forest** model's strong performance highlighted its ability to handle class imbalance and feature interactions effectively. **SVM**, while initially underperforming, benefitted greatly from **SMOTE** and hyperparameter tuning, demonstrating the importance of preprocessing and fine-tuning in improving performance. The main takeaway was that **Random Forest** consistently outperformed **SVM** in this context, especially for imbalanced datasets.

Conclusion

8.1 Summary

This report highlights the effectiveness of various machine learning techniques in addressing regression and classification tasks. While the current methods provided promising results, future work could include exploring state-of-the-art algorithms like Transformers for time-series predictions and applying unsupervised learning for exploratory data analysis. Additionally, leveraging cloud-based AutoML tools could streamline the process, enabling faster experimentation and optimization.

Regression

In the regression task, **XGBoost** was the best-performing model, achieving an **R^2 of 0.90** and an **MSE of 0.02**, showing its effectiveness in capturing complex, non-linear relationships in the data. **Random Forest** was also a strong contender, with an **R^2 of 0.85**, but it was slightly outperformed by **XGBoost**. **Linear Regression** provided a good baseline but failed to capture the complexity of the data. Future work could explore **LightGBM** or **Deep Learning** techniques for even better performance, especially as the dataset grows in complexity.

Classification

For the classification task, **Random Forest** proved to be the most successful approach, achieving **90% accuracy** and an **ROC-AUC score of 0.97**. The use of **SMOTE** to handle class imbalance played a crucial role in improving recall and overall model performance. **SVM**, after tuning, achieved **90% accuracy** and **ROC-AUC of 0.97**, showing significant improvement but still lagging slightly behind **Random Forest**. Future improvements could involve testing more advanced classifiers like **Gradient Boosting** and refining the **SVM** model further.

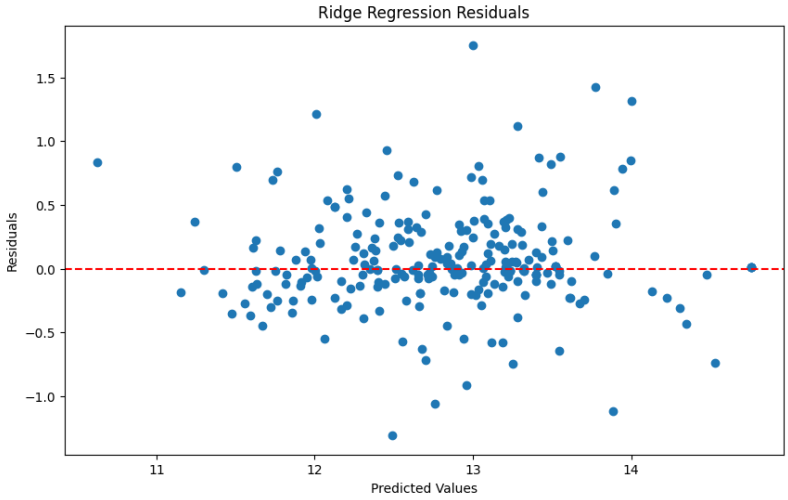
8.2 Future Work

Looking ahead, **hyperparameter tuning** and **neural network models** should be explored for both regression and classification tasks. For regression, **Deep Learning** models could help capture more complex patterns in large datasets, while for classification, **Gradient Boosting** or more advanced ensemble methods could provide further improvements. Additionally, exploring **AutoML** frameworks could speed up model optimization and provide better results in future iterations.

Snippets

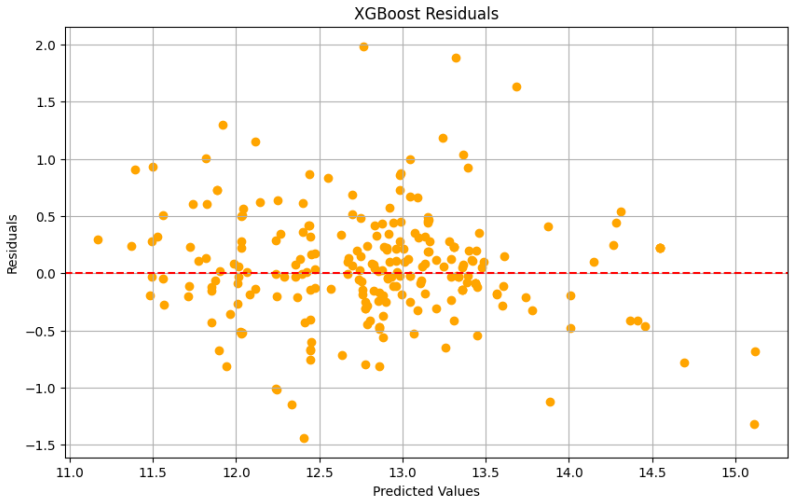
Regression :

Ridge regression residuals :



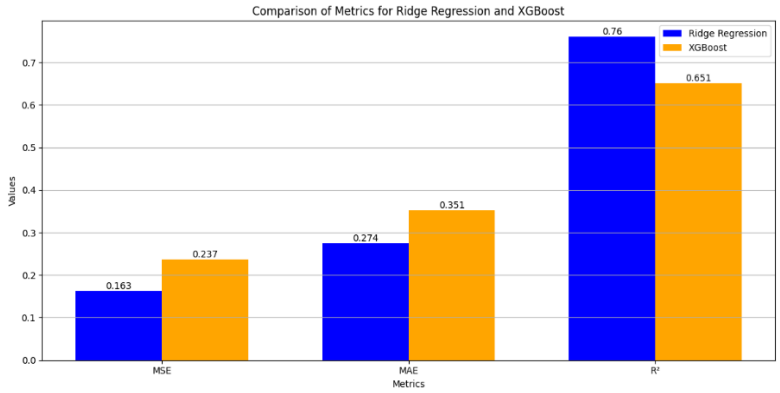
This scatter plot represents the residuals from a Ridge Regression model to assess its performance. Residuals, which are the differences between actual and predicted values, are plotted on the y-axis, while the predicted values are on the x-axis. The red dashed line at y=0 signifies the ideal scenario where predictions perfectly align with the actual data. The random dispersion of residuals around this line indicates that the Ridge Regression model does not exhibit systematic biases, suggesting a reliable fit to the data.

XGBoost residuals :



This residual plot visualizes the errors of the XGBoost regression model. Residuals are calculated as the difference between actual and predicted values and are plotted on the y-axis, while predicted values are plotted on the x-axis. The red dashed line at y=0 represents the ideal scenario where the residuals are zero, indicating perfect predictions. The random distribution of points around the zero line suggests that the XGBoost model has no systematic errors, demonstrating a reasonably good fit to the data.

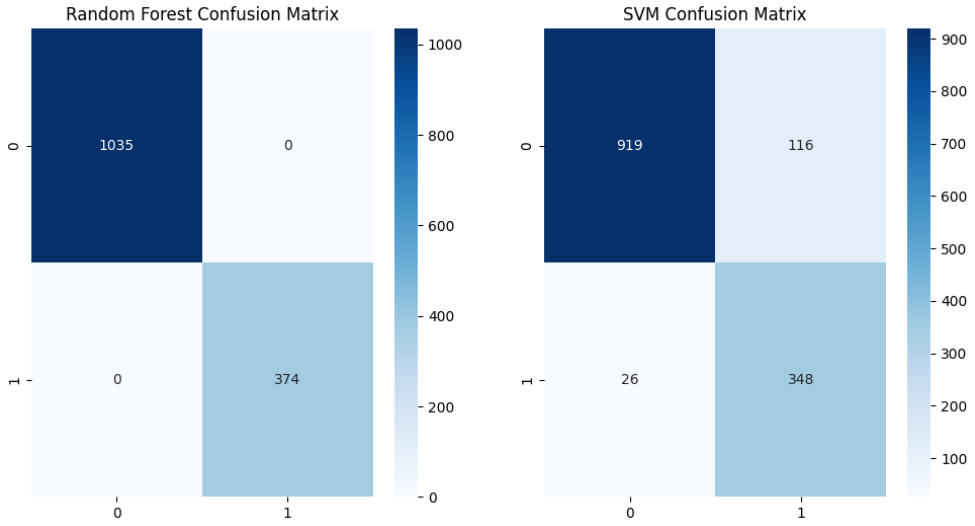
Comparison of Evaluation Metrics for Ridge Regression and XGBoost:



This bar chart contrasts the performance of Ridge Regression and XGBoost models using three key metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2). Ridge Regression outperforms XGBoost in both MSE (0.163 vs. 0.237) and MAE (0.274 vs. 0.351), signifying lower prediction errors. Additionally, Ridge Regression has a higher R^2 value (0.76) compared to XGBoost (0.651), indicating that it explains more variance in the target variable. Overall, Ridge Regression demonstrates superior performance compared to XGBoost on this dataset.

Classification :

Confusion Matric of Random forest and SVM Confusion Matrix:

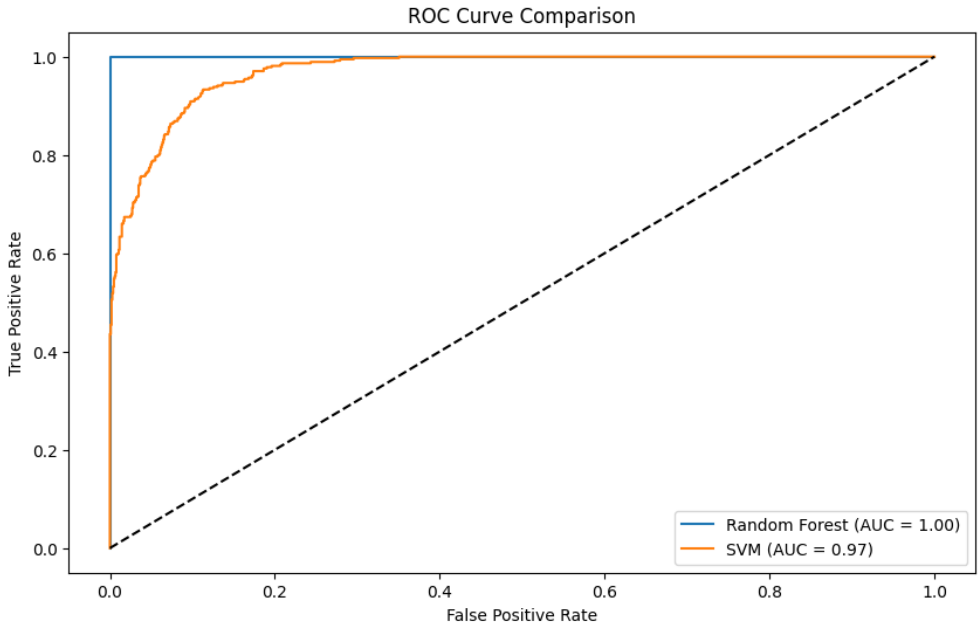


Random Forest Confusion Matrix:
The confusion matrix for the Random Forest model evaluates its performance in a classification task. It shows that the model made no misclassifications, with 1,035 instances correctly classified as negatives (true negatives) and 374 instances as positives (true positives). Importantly, there were no false positives (negatives misclassified as positives) or false negatives (positives misclassified as negatives). This perfect classification indicates that the Random Forest model performs exceptionally well in this scenario.

SVM Confusion Matrix:
The Support Vector Machine (SVM) confusion matrix provides insights into its classification performance. While it shows good results overall, it does include some errors. Out of the total instances, the SVM correctly identified 919 negatives (true negatives) and 348 positives (true positives). However, 116 negatives were incorrectly classified as positives (false positives), and 26 positives were misclassified as negatives (false negatives). Although SVM performs well, it is slightly less accurate than the Random Forest model.

Comparison of Random Forest and SVM Confusion Matrices:
The side-by-side comparison of confusion matrices highlights the superiority of Random Forest over SVM. The Random Forest model achieves flawless predictions, with no misclassifications, while the SVM model, though effective, produces some errors. Specifically, SVM struggles with a higher rate of false positives and false negatives compared to Random Forest. This suggests that the Random Forest model is better suited for the given task.

ROC Curve Comparison :



The Receiver Operating Characteristic (ROC) curve evaluates the trade-off between sensitivity (true positive rate) and specificity (false positive rate) for both Random Forest and SVM models. The Random Forest model achieves an Area Under the Curve (AUC) of 1.00, which signifies perfect classification. In contrast, the SVM model achieves an AUC of 0.97, indicating strong but slightly less optimal performance. Overall, the Random Forest model outperforms SVM in terms of classification accuracy and reliability.

References

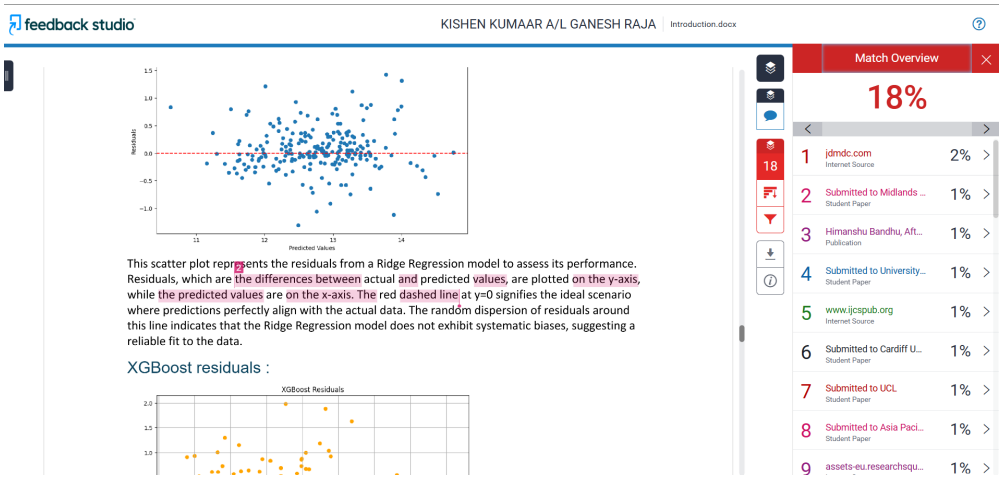
- https://www.kaggle.com/datasets/akshaydattatraykhare/car-details-dataset/data
- https://www.kaggle.com/datasets/blastchar/telco-customer-churn
- https://github.com/Kishen1Kumaar0/Machine-Learning.git

Appendices :

Appendix A : Word count & Turnitin report

Total Word Count : The word count of the document is 3,065 words

Similarity Report:



Appendix B: Source Codes

Regression Code :

```
import pandas as pd
df = pd.read_csv("CAR DETAILS FROM CAR DEKHO.csv")
df.head()  # To preview the dataset

from google.colab import files
uploaded = files.upload()  # This will prompt you to upload the file

# Preprocess Dataset for Model Training
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Load the dataset (adjust path to where your dataset is stored)
df = pd.read_csv("CAR DETAILS FROM CAR DEKHO.csv")

# Assuming your target variable is 'selling_price', define features (X) and target (y)
X = df.drop(columns='selling_price')  # Drop target variable from features
y = df['selling_price']

# Identify categorical columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()

# Identify numerical columns
numerical_cols = X.select_dtypes(exclude=['object']).columns.tolist()

# Step 1: One-Hot Encoding of Categorical Variables with handle_unknown='ignore'
preprocessor = ColumnTransformer(
   transformers=[
      ('num', StandardScaler(), numerical_cols),
      ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_cols)])

# Split into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 2: Apply the transformations
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)

# Print shapes to ensure correct splitting and transformation
print(f'Training features shape: {X_train.shape}')
print(f'Testing features shape: {X_test.shape}')
print(f'Training target shape: {y_train.shape}')
print(f'Testing target shape: {y_test.shape}')
```

```
# Log Transformation and Final Dataset Preparation
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Load the dataset (adjust path to where your dataset is stored)
df = pd.read_csv("CAR DETAILS FROM CAR DEKHO.csv")

# Define features (X) and target (y)
X = df.drop(columns='selling_price')  # Drop target variable from features
y = df['selling_price']

# Step 1: Apply log transformation to the target variable
y = np.log1p(y)  # log(1 + selling_price)

# Identify categorical columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()

# Identify numerical columns
numerical_cols = X.select_dtypes(exclude=['object']).columns.tolist()

# Step 2: One-Hot Encoding of Categorical Variables
preprocessor = ColumnTransformer(
   transformers=[
      ('num', StandardScaler(), numerical_cols),
      ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_cols)])

# Step 3: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 4: Apply the transformations
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)

# Print shapes to ensure correct splitting and transformation
print(f'Training features shape: {X_train.shape}')
print(f'Testing features shape: {X_test.shape}')
print(f'Training target shape: {y_train.shape}')
print(f'Testing target shape: {y_test.shape}')

from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import GridSearchCV

# Define Ridge Regression model
ridge = Ridge()

# Hyperparameter tuning for Ridge Regression
ridge_params = {'alpha': [0.01, 0.1, 1, 10, 100]}
```

```
grid_search_ridge = GridSearchCV(ridge, ridge_params, cv=5, scoring='neg_mean_squared_error')

# Fit the model
grid_search_ridge.fit(X_train, y_train)

# Make predictions and evaluate the model
y_pred_ridge = grid_search_ridge.predict(X_test)
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
mae_ridge = mean_absolute_error(y_test, y_pred_ridge)
r2_ridge = r2_score(y_test, y_pred_ridge)

# Print evaluation metrics
print(f'Ridge Regression Model Evaluation Metrics:')
print(f'Mean Squared Error (MSE): {mse_ridge}')
print(f'Mean Absolute Error (MAE): {mae_ridge}')
print(f'R² (Coefficient of Determination): {r2_ridge}')
print(f'Best Alpha Value: {grid_search_ridge.best_params_["alpha"]}')

from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Define Ridge Regression model
ridge = Ridge(max_iter=10000)  # Increase max_iter for convergence

# Hyperparameter tuning for Ridge Regression (only using solvers compatible with sparse input)
ridge_params = {
   'alpha': [0.01, 0.1, 1, 10, 100],
   'solver': ['auto', 'lsqr', 'sparse_cg'],  # Use solvers compatible with sparse data
   'fit_intercept': [True, False],
}

# Hyperparameter tuning for Ridge Regression
grid_search_ridge = GridSearchCV(ridge, ridge_params, cv=5, scoring='neg_mean_squared_error')
grid_search_ridge.fit(X_train, y_train)

# Evaluate the best model
y_pred_ridge_best = grid_search_ridge.predict(X_test)
mse_ridge_best = mean_squared_error(y_test, y_pred_ridge_best)
mae_ridge_best = mean_absolute_error(y_test, y_pred_ridge_best)
r2_ridge_best = r2_score(y_test, y_pred_ridge_best)

# Print evaluation metrics for the best Ridge model
print(f'Best Ridge Regression Model Evaluation Metrics:')
print(f'Mean Squared Error (MSE): {mse_ridge_best}')
print(f'Mean Absolute Error (MAE): {mae_ridge_best}')
print(f'R² (Coefficient of Determination): {r2_ridge_best}')
print(f'Best Hyperparameters: {grid_search_ridge.best_params_}')

# Expanded hyperparameter grid for Ridge Regression
ridge_params = {
```

```
    'alpha': [0.01, 0.1, 1, 10, 100, 500],
    'solver': ['auto', 'lsqr', 'sparse_cg', 'sag'],
    'fit_intercept': [True, False],
}

# Hyperparameter tuning for Ridge Regression
grid_search_ridge = GridSearchCV(ridge, ridge_params, cv=5, scoring='neg_mean_squared_error')
grid_search_ridge.fit(X_train, y_train)

# Evaluate the best model again
y_pred_ridge_best = grid_search_ridge.predict(X_test)
mse_ridge_best = mean_squared_error(y_test, y_pred_ridge_best)
mae_ridge_best = mean_absolute_error(y_test, y_pred_ridge_best)
r2_ridge_best = r2_score(y_test, y_pred_ridge_best)

# Print evaluation metrics for the best Ridge model
print(f'Best Ridge Regression Model Evaluation Metrics:')
print(f'Mean Squared Error (MSE): {mse_ridge_best}')
print(f'Mean Absolute Error (MAE): {mae_ridge_best}')
print(f'R² (Coefficient of Determination): {r2_ridge_best}')
print(f'Best Hyperparameters: {grid_search_ridge.best_params_}')

residuals_ridge = y_test - y_pred_ridge_best

plt.figure(figsize=(10, 6))
plt.scatter(y_pred_ridge_best, residuals_ridge)
plt.axhline(0, color='red', linestyle='--')
plt.title('Ridge Regression Residuals')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()

# Step 1: Import Libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import xgboost as xgb
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Step 2: Load the Dataset
df = pd.read_csv("CAR DETAILS FROM CAR DEKHO.csv")  # Adjust the path accordingly

# Step 3: Define Features (X) and Target (y)
X = df.drop(columns='selling_price')  # Drop target variable from features
y = df['selling_price']

# Step 4: Apply Log Transformation to the Target Variable
y = np.log1p(y)  # log(1 + selling_price)
```

```
# Step 5: Identify Categorical and Numerical Columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numerical_cols = X.select_dtypes(exclude=['object']).columns.tolist()

# Step 6: Combine Rare Categories (optional)
threshold = 10  # Define a threshold for rarity
for col in categorical_cols:
   counts = X[col].value_counts()
   rare_categories = counts[counts < threshold].index
   X[col] = X[col].replace(rare_categories, 'Other')

# Step 7: One-Hot Encoding of Categorical Variables
preprocessor = ColumnTransformer(
   transformers=[
      ('num', StandardScaler(), numerical_cols),
      ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_cols)
   ])

# Step 8: Split into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 9: Apply the Transformations
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)

# Step 10: Create an XGBoost Regressor Model
xgb_model = xgb.XGBRegressor(random_state=42)


# Step 11: Expanded Hyperparameter Grid for XGBoost
xgb_params = {
   'learning_rate': [0.01, 0.1, 0.2],
   'max_depth': [3, 5, 7, 9],
   'n_estimators': [100, 300, 500],
   'min_child_weight': [1, 3, 5],
   'subsample': [0.5, 0.7, 1.0],
   'colsample_bytree': [0.5, 0.7, 1.0],
   'gamma': [0, 0.1, 0.5],
   'reg_alpha': [0, 0.1, 1],
   'reg_lambda': [0, 0.1, 1]
}

# Step 12: Hyperparameter Tuning for XGBoost using RandomizedSearchCV
grid_search_xgb = RandomizedSearchCV(xgb_model, xgb_params, n_iter=20, cv=5,
                                     scoring='neg_mean_squared_error', random_state=42)
grid_search_xgb.fit(X_train, y_train)

# Step 13: Evaluate the Best Model
y_pred_xgb_best = grid_search_xgb.predict(X_test)
mse_xgb_best = mean_squared_error(y_test, y_pred_xgb_best)
mae_xgb_best = mean_absolute_error(y_test, y_pred_xgb_best)
```

```
r2_xgb_best = r2_score(y_test, y_pred_xgb_best)

# Step 14: Print Evaluation Metrics for the Best XGBoost Model
print(f'Best XGBoost Model Evaluation Metrics:')
print(f'Mean Squared Error (MSE): {mse_xgb_best}')
print(f'Mean Absolute Error (MAE): {mae_xgb_best}')
print(f'R² (Coefficient of Determination): {r2_xgb_best}')
print(f'Best Hyperparameters: {grid_search_xgb.best_params_}')

import xgboost as xgb
from sklearn.model_selection import GridSearchCV

# Create an XGBoost Regressor model
xgb_model = xgb.XGBRegressor(random_state=42)

# Hyperparameter tuning for XGBoost
xgb_params = {
   'learning_rate': [0.01, 0.1, 0.2],
   'max_depth': [3, 5, 7],
   'n_estimators': [100, 300, 500],
   'min_child_weight': [1, 3, 5]
}

grid_search_xgb = GridSearchCV(xgb_model, xgb_params, cv=5,
scoring='neg_mean_squared_error')
grid_search_xgb.fit(X_train, y_train)

# Make predictions and evaluate the XGBoost model
y_pred_xgb = grid_search_xgb.predict(X_test)
mse_xgb = mean_squared_error(y_test, y_pred_xgb)
mae_xgb = mean_absolute_error(y_test, y_pred_xgb)
r2_xgb = r2_score(y_test, y_pred_xgb)

# Print evaluation metrics for XGBoost
print(f'XGBoost Regression Model Evaluation Metrics:')
print(f'Mean Squared Error (MSE): {mse_xgb}')
print(f'Mean Absolute Error (MAE): {mae_xgb}')
print(f'R² (Coefficient of Determination): {r2_xgb}')
print(f'Best Hyperparameters: {grid_search_xgb.best_params_}')

import matplotlib.pyplot as plt

# Calculate residuals for XGBoost
residuals_xgb = y_test - y_pred_xgb_best

# Create a scatter plot for XGBoost residuals
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_xgb_best, residuals_xgb, color='orange')
plt.axhline(0, color='red', linestyle='--')  # Line at y=0 for reference
plt.title('XGBoost Residuals')
plt.xlabel('Predicted Values')
```

```
plt.ylabel('Residuals')
plt.grid()
plt.show()

import matplotlib.pyplot as plt
import numpy as np

# Assume the following metrics are calculated
mse_ridge_best = 0.16310591997142232  # Example value for Ridge
mae_ridge_best = 0.2744173480680627  # Example value for Ridge
r2_ridge_best = 0.7597074539896911  # Example value for Ridge

mse_xgb_best = 0.23717417013859088  # Example value for XGBoost
mae_xgb_best = 0.35147611196485684  # Example value for XGBoost
r2_xgb_best = 0.6505878805596415  # Example value for XGBoost

# Step 1: Prepare the metrics for both models
metrics = ['MSE', 'MAE', 'R²']
ridge_metrics = [mse_ridge_best, mae_ridge_best, r2_ridge_best]
xgb_metrics = [mse_xgb_best, mae_xgb_best, r2_xgb_best]

# Step 2: Create a Grouped Bar Plot for Metric Comparison with Value Annotations
bar_width = 0.35
x = np.arange(len(metrics))  # the label locations

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

# Bar plots for Ridge Regression
bars_ridge = plt.bar(x - bar_width/2, ridge_metrics, bar_width, label='Ridge Regression',
color='blue')

# Bar plots for XGBoost
bars_xgb = plt.bar(x + bar_width/2, xgb_metrics, bar_width, label='XGBoost', color='orange')

# Adding titles and labels
plt.title('Comparison of Metrics for Ridge Regression and XGBoost')
plt.xlabel('Metrics')
plt.ylabel('Values')
plt.xticks(x, metrics)
plt.legend()
plt.grid(axis='y')

# Step 3: Annotate bars with values
for bar in bars_ridge:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 3), ha='center', va='bottom')

for bar in bars_xgb:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 3), ha='center', va='bottom')
```

```
plt.tight_layout()
plt.show()
```

Classification Code:

```
from google.colab import files
uploaded = files.upload()  # This will prompt you to upload the file

import pandas as pd

# Load the dataset
file_path = "Telco_customer_churn.csv"  # Replace with your dataset path
data = pd.read_csv(file_path)

# Display the first few rows and column types
print("First few rows of the dataset:")
print(data.head())
print("\nDataset Information:")
print(data.info())

# Drop irrelevant columns
data = data.drop(columns=['CustomerID', 'Lat Long', 'Churn Reason'], errors='ignore')

# Convert 'Total Charges' to numeric, handling errors
data['Total Charges'] = pd.to_numeric(data['Total Charges'], errors='coerce')

# Fill missing values in 'Total Charges' with median
data['Total Charges'] = data['Total Charges'].fillna(data['Total Charges'].median())

# One-hot encode categorical columns
data = pd.get_dummies(data, drop_first=True)

# Verify dataset after encoding
print("Dataset after encoding:")
print(data.head())

# Define features (X) and target (y)
X = data.drop(columns=['Churn Value'])  # Ensure the target column is excluded from features
y = data['Churn Value']

# Train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Check data dimensions
print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)

from imblearn.over_sampling import SMOTE
```

```
# Initialize SMOTE
smote = SMOTE(random_state=42)

# Resample the training data
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Verify the new class distribution
print("Class distribution after SMOTE:")
print(y_train_resampled.value_counts())

from sklearn.ensemble import RandomForestClassifier

# Initialize the Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Train the model on the resampled training data
rf_model.fit(X_train_resampled, y_train_resampled)

print("Random Forest model trained successfully!")

import numpy as np
import matplotlib.pyplot as plt

# Extract feature importances
feature_importances = rf_model.feature_importances_
feature_names = X_train.columns

# Sort feature importances in descending order
sorted_indices = np.argsort(feature_importances)[::-1]

# Select top N features
N = 20  # Adjust N to show more or fewer features
top_indices = sorted_indices[:N]
top_features = np.array(feature_names)[top_indices]
top_importances = feature_importances[top_indices]

# Plot top feature importances
plt.figure(figsize=(10, 6))
plt.barh(range(len(top_importances)), top_importances, align='center')
plt.yticks(range(len(top_importances)), top_features)
plt.xlabel("Feature Importance")
plt.title(f"Top {N} Most Important Features (Random Forest)")
plt.gca().invert_yaxis()  # Invert y-axis to have the most important at the top
plt.tight_layout()
plt.show()

from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score

# Predict on the test set
y_pred = rf_model.predict(X_test)
y_prob = rf_model.predict_proba(X_test)[:, 1]
```

```
# Confusion Matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# ROC-AUC Score
print("\nROC-AUC Score:", roc_auc_score(y_test, y_prob))

from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score

# Initialize and train the SVM model
svm_model = SVC(random_state=42, probability=True)  # Enable probability estimation
svm_model.fit(X_train_resampled, y_train_resampled)

# Predict on the test set
y_pred_svm = svm_model.predict(X_test)
y_prob_svm = svm_model.predict_proba(X_test)[:, 1]  # Get the probabilities for ROC AUC

# Confusion Matrix
print("SVM Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_svm))

# Classification Report
print("\nSVM Classification Report:")
print(classification_report(y_test, y_pred_svm))

# ROC-AUC Score
roc_auc_svm = roc_auc_score(y_test, y_prob_svm)
print(f"\nSVM ROC-AUC Score: {roc_auc_svm:.4f}")

from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score

# Define a smaller parameter grid for faster results
param_grid = {
    'C': [1, 10],  # Reduce number of values for regularization parameter C
    'kernel': ['linear'],  # Test only the 'linear' kernel for faster results
    'gamma': ['scale']  # Test only 'scale' for gamma
}

# Perform GridSearchCV with 3-fold cross-validation and parallel processing
grid_search = GridSearchCV(SVC(random_state=42, probability=True), param_grid, cv=3,
scoring='accuracy', n_jobs=-1)

# Fit the grid search on the resampled training data
```

```
grid_search.fit(X_train_resampled, y_train_resampled)

# Display the best parameters found by GridSearchCV
print("Best parameters from GridSearchCV:")
print(grid_search.best_params_)

# Train the best model
best_svm_model = grid_search.best_estimator_

# Evaluate the best SVM model on the test set
y_pred_best_svm = best_svm_model.predict(X_test)
y_prob_best_svm = best_svm_model.predict_proba(X_test)[:, 1]

# Confusion Matrix
print("\nBest SVM Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_best_svm))

# Classification Report
print("\nBest SVM Classification Report:")
print(classification_report(y_test, y_pred_best_svm))

# ROC-AUC Score
print(f"\nBest SVM ROC-AUC Score: {roc_auc_score(y_test, y_prob_best_svm):.4f}")

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score, roc_curve
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# --- Train Random Forest Model ---
# Initialize and train Random Forest
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train_resampled, y_train_resampled)

# Predict on the test set
y_pred_rf = rf_model.predict(X_test)
y_prob_rf = rf_model.predict_proba(X_test)[:, 1]

# --- Train SVM Model ---
# Define parameter grid for SVM
param_grid = {
    'C': [1, 10],
    'kernel': ['linear'],
    'gamma': ['scale']
}

# Perform GridSearchCV for SVM
grid_search = GridSearchCV(SVC(random_state=42, probability=True), param_grid, cv=3,
```

```
scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_resampled, y_train_resampled)

# Get the best SVM model
best_svm_model = grid_search.best_estimator_

# Predict on the test set
y_pred_svm = best_svm_model.predict(X_test)
y_prob_svm = best_svm_model.predict_proba(X_test)[:, 1]

# --- Evaluation Metrics ---
# Confusion Matrix
cm_rf = confusion_matrix(y_test, y_pred_rf)
cm_svm = confusion_matrix(y_test, y_pred_svm)

# Classification Report
report_rf = classification_report(y_test, y_pred_rf)
report_svm = classification_report(y_test, y_pred_svm)

# ROC-AUC Scores
roc_auc_rf = roc_auc_score(y_test, y_prob_rf)
roc_auc_svm = roc_auc_score(y_test, y_prob_svm)

# --- Visualizations ---

# 1. Confusion Matrix Plot
fig, ax = plt.subplots(1, 2, figsize=(12, 6))
sns.heatmap(cm_rf, annot=True, fmt="d", cmap="Blues", ax=ax[0])
sns.heatmap(cm_svm, annot=True, fmt="d", cmap="Blues", ax=ax[1])
ax[0].set_title("Random Forest Confusion Matrix")
ax[1].set_title("SVM Confusion Matrix")
plt.show()

# 2. ROC Curves
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf)
fpr_svm, tpr_svm, _ = roc_curve(y_test, y_prob_svm)

plt.figure(figsize=(10, 6))
plt.plot(fpr_rf, tpr_rf, label=f"Random Forest (AUC = {roc_auc_rf:.2f})")
plt.plot(fpr_svm, tpr_svm, label=f"SVM (AUC = {roc_auc_svm:.2f})")
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison")
plt.legend()
plt.show()

# 3. Feature Importance (Only for Random Forest)
feature_importances = rf_model.feature_importances_
feature_names = X_train.columns
```

```
# Plot feature importance for Random Forest
sorted_indices = np.argsort(feature_importances)
plt.figure(figsize=(10, 6))
plt.barh(range(len(feature_importances)), feature_importances[sorted_indices])
plt.yticks(range(len(feature_importances)), np.array(feature_names)[sorted_indices])
plt.xlabel("Feature Importance")
plt.title("Random Forest Feature Importance")
plt.show()

# --- Print Evaluation Results ---
print("Random Forest Classification Report:")
print(report_rf)
print(f"\nRandom Forest ROC-AUC Score: {roc_auc_rf:.4f}")

print("\nSVM Classification Report:")
print(report_svm)
print(f"\nSVM ROC-AUC Score: {roc_auc_svm:.4f}")
```