School of IT & Business Technologies Graduate Diploma in Data Analytics

Course Title:	Machine Learning and AI	**Course code:**	GDDA708
Student Name:	HARIPRASANTH SINNVAJOUMOUNY	**Student ID:**	764707774
Assessment No & Type:	Assessment 2[Portfolio]	**Cohort:**	GDDA7123C
Due Date:	7/3/2024	**Date Submitted:**	29/03/2024
Tutor's Name:	Dr, Harsh Tiwari		
Assessment Weighting	60%		
Total Marks	100		

Student Declaration:

I declare that:

- I have read the New Zealand School of Education Ltd policies and regulations on assessments and understand what plagiarism is.
- I am aware of the penalties for cheating and plagiarism as laid down by the New Zealand School of Education Ltd.
- This is an original assessment and is entirely my own work.
- Where I have quoted or made use of the ideas of other writers, I have acknowledged the source.
- This assessment has been prepared exclusively for this course and has not been or will not be submitted as assessed work in any other course.
- It has been explained to me that this assessment may be used by NZSE Ltd, for internal and/or external moderation.

- If I am late in handing in this assessment without prior approval (see student regulations in handbook), marks will be deducted, to a maximum of 50%.

Student signature:

Date: 29/03/2024

Tutor only to complete		
Assessment result:	**Mark /100**	**Grade**

Contents

PART A – Regression modelling for business decision making

Introduction

In this analysis, we embark on a journey to unveil insights from a dataset centered around insurance costs. Given the dataset's potential to inform critical business decisions, our objective is to meticulously prepare the data, ensuring its readiness for regression analysis. This process encompasses cleaning the dataset, handling missing values, removing outliers, and ultimately, building and evaluating regression models to predict insurance charges.

a) Cleaning the Dataset, Handling Missing Values, and Removing Outliers

Loading the Dataset

Our analytical endeavor begins with the importation of the dataset into our analysis environment. This preliminary step is pivotal, offering us a snapshot of the data we will engage with.

Task 1 (a): Cleaning the Dataset, Handling Missing Values, and Removing Outliers

In [9]:
```
# Import pandas library for data manipulation
import pandas as pd

# Load the dataset into a pandas DataFrame
df = pd.read_csv("C:\\Users\\harip\OneDrive\\Desktop\\Data analytics\\machine leaning\\assessment1\\dataset\\expenses.csv")


# Display the first few rows to get an overview of the data
initial_view = df.head()
initial_view
```

Out[9]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

The dataset is successfully imported, providing an initial glimpse into its structure and contents. This early inspection is crucial for informing our preprocessing strategy.

Handling Missing Values

The integrity of our dataset is paramount. As such, our next step involves identifying and addressing missing values within the dataset.

In [10]:
```
# Check for missing values in each column
missing_values = df.isnull().sum()

# Display the missing values count per column
missing_values_count = missing_values[missing_values > 0]
missing_values_count  # Display the variable to see the missing values in the report

# If missing values are found, opt to remove these rows
if missing_values_count.any():
    df = df.dropna()

# Display the shape of the dataset after removing missing values
post_missing_values_removal_shape = df.shape
post_missing_values_removal_shape  # Display the variable to see the new shape
```

Out[10]: (1338, 7)

A thorough examination reveals missing values within our dataset. In our pursuit of data integrity and to prevent potential biases that imputation might introduce, we have opted to exclude entries lacking complete information

Removing Outliers

As part of our data preparation, we focused on the crucial task of identifying data points that stand out excessively from the rest, commonly known as outliers. These outliers can have a disproportionate effect on our model's performance and may lead to biased results.

To address this, we implemented a statistical technique involving Z-scores, which essentially helps us understand how far a data point is from the mean in terms of standard deviations.

In [15]:
```
from scipy import stats
import numpy as np

# Calculate the Z-score for each data point
z_scores = stats.zscore(df.select_dtypes(include=[np.number]))
abs_z_scores = np.abs(z_scores)

# Define a threshold for identifying outliers
threshold = 3

# Remove outliers
df_clean = df[(abs_z_scores < threshold).all(axis=1)]

# Display the shape of the dataset after removing outliers
post_outliers_removal_shape = df_clean.shape
post_outliers_removal_shape  # Display the variable to see the new shape after outlier removal
```

Out[15]: (1309, 7)

In our analysis, we used a threshold of three standard deviations, beyond which data points are deemed outliers and thus excluded from the dataset. This careful and deliberate process of outlier removal is essential to ensuring the accuracy and generalizability of our model.

b) Standardization of Numerical Features

For the integrity of our regression analysis, scaling numerical features is a pivotal step that we cannot overlook. It ensures that each feature contributes proportionately to the final prediction, eliminating any bias that could arise from the raw scale of the data.

To execute this step efficiently, we utilized the StandardScaler from the sklearn library.

b) Feature Scaling and Normalization

In [16]:
```
from sklearn.preprocessing import StandardScaler

# Assuming df_clean is your DataFrame after removing missing values and outliers

# Initialize the StandardScaler
scaler = StandardScaler()

# List of numerical features to scale (excluding 'charges' as it is the target variable)
numerical_features = ['age', 'bmi', 'children']

# Create a copy of the DataFrame to avoid the SettingWithCopyWarning
# and apply scaling to the numerical features directly
df_clean.loc[:, numerical_features] = scaler.fit_transform(df_clean[numerical_features])

# Display the scaled features to verify the scaling
scaled_features_preview = df_clean.head()
print(scaled_features_preview)
```

```
        age     sex       bmi  children smoker     region      charges
0 -1.439063  female -0.449359 -0.929616    yes  southwest  16884.92400
1 -1.510086    male  0.533391 -0.040093     no  southeast   1725.55230
2 -0.799859    male  0.404478  1.738954     no  southeast   4449.46200
3 -0.444746    male -1.319101 -0.929616     no  northwest  21984.47061
4 -0.515769    male -0.285288 -0.929616     no  northwest   3866.85520
```

Through this code snippet, we've standardized the features 'age', 'bmi', and 'children' to have a mean of zero and a standard deviation of one. This transformation is a cornerstone for many machine learning algorithms that are sensitive to the magnitude of inputs.

The resulting DataFrame, now with scaled numerical features, stands ready for the predictive modeling we are set to undertake. By normalizing these variables, we set a level playing field, ensuring that no single feature will unduly influence the model's outcome due to its scale.

c) Categorical Variable Transformation for Regression Analysis

In our pursuit of building a predictive model for insurance costs, we encountered categorical variables 'sex', 'smoker', and 'region'. To seamlessly integrate these into our regression model, we applied a technique known as One-Hot Encoding.

In [17]:

```
from sklearn.preprocessing import OneHotEncoder

# Initialize OneHotEncoder
encoder = OneHotEncoder(sparse=False)

# Select categorical data
categorical_features = ['sex', 'smoker', 'region']

# Assuming df_clean is your DataFrame after scaling numerical features
# Fit the encoder and transform the categorical data
encoded_features = encoder.fit_transform(df_clean[categorical_features])

# Create a DataFrame with the encoded variables
encoded_vars_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(categorical_features))

# Concatenate the encoded variables with the original DataFrame
# Drop original categorical features to avoid redundancy
df_final = pd.concat([df_clean.drop(categorical_features, axis=1).reset_index(drop=True), encoded_vars_df.reset_index(drop=Tr

# Display the DataFrame to verify the encoding
encoded_features_preview = df_final.head()
print(encoded_features_preview)
```

```
        age       bmi  children      charges  sex_female  sex_male  smoker_no  \
0 -1.439063 -0.449359 -0.929616  16884.92400         1.0       0.0        0.0
1 -1.510086  0.533391 -0.040093   1725.55230         0.0       1.0        1.0
2 -0.799859  0.404478  1.738954   4449.46200         0.0       1.0        1.0
3 -0.444746 -1.319101 -0.929616  21984.47061         0.0       1.0        1.0
4 -0.515769 -0.285288 -0.929616   3866.85520         0.0       1.0        1.0

   smoker_yes  region_northeast  region_northwest  region_southeast  \
0         1.0               0.0               0.0               0.0
1         0.0               0.0               0.0               1.0
2         0.0               0.0               0.0               1.0
3         0.0               0.0               1.0               0.0
4         0.0               0.0               1.0               0.0

   region_southwest
0               1.0
```

This process converts categorical variables into a numerical format that our regression algorithms can interpret. Each category is represented by a new binary column, indicating its presence or absence for each record. It's a critical step, as it converts human-readable categories into a machine-friendly format, facilitating the uncovering of patterns related to the costs associated with insurance policies.

d) Splitting the Dataset into Training and Testing Sets

To evaluate our models' performance accurately, we partition our dataset into training and testing sets. This strategy enables us to train our models on one subset of the data and validate their performance on a separate, unseen subset.

In [18]:
```
from sklearn.model_selection import train_test_split

# Assuming df_final is your preprocessed DataFrame
# Define the features (X) and the target (y). Here, 'charges' is considered as the target variable.
X = df_final.drop('charges', axis=1)
y = df_final['charges']

# Split the data into training and testing sets. Here, we'll use 80% of the data for training and 20% for testing.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Display the shapes of the training and testing sets to verify the split
print("Training set shape (X_train):", X_train.shape)
print("Testing set shape (X_test):", X_test.shape)
print("Training target shape (y_train):", y_train.shape)
print("Testing target shape (y_test):", y_test.shape)
```

```
Training set shape (X_train): (1047, 11)
Testing set shape (X_test): (262, 11)
Training target shape (y_train): (1047,)
Testing target shape (y_test): (262,)
```

We have allocated 80% of our dataset for training and reserved 20% for testing. This split ensures that we have sufficient data to train our models while also holding back a portion for an unbiased evaluation of their performance.

Task 2: Model Building, Hyperparameter Tuning, and Initial Evaluation

a) Building and Evaluating Linear Regression and Random Forest Models

We proceed to build two types of regression models: **Linear Regression and Random Forest**. These models are selected for their ability to capture both linear relationships and complex, nonlinear interactions within the data.

Linear Regression Model:

 # Building a Linear Regression Model

In [20]:
```
from sklearn.linear_model import LinearRegression

# Initialize the Linear Regression model
lr_model = LinearRegression()

# Fit the model on the training data
lr_model.fit(X_train, y_train)

# Make predictions on the testing set
y_pred_lr = lr_model.predict(X_test)

# Evaluate the model's performance
mae_lr = mean_absolute_error(y_test, y_pred_lr)
mse_lr = mean_squared_error(y_test, y_pred_lr)
r2_lr = r2_score(y_test, y_pred_lr)

print("Linear Regression Performance:")
print("Mean Absolute Error:", mae_lr)
print("Mean Squared Error:", mse_lr)
print("R^2 Score:", r2_lr)
```

```
Linear Regression Performance:
Mean Absolute Error: 3969.0341267772073
Mean Squared Error: 30444091.526395813
R^2 Score: 0.7787373616773112
```

Random Forest Model:

Building a Random Forest Regressor Model

In [19]:
```
from sklearn.ensemble import RandomForestRegressor

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

# Fit the model on the training data
rf_model.fit(X_train, y_train)

# Make predictions on the testing set
y_pred_rf = rf_model.predict(X_test)

# Evaluate the model's performance
mae_rf = mean_absolute_error(y_test, y_pred_rf)
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print("Random Forest Regressor Performance:")
print("Mean Absolute Error:", mae_rf)
print("Mean Squared Error:", mse_rf)
print("R^2 Score:", r2_rf)
```

```
Random Forest Regressor Performance:
Mean Absolute Error: 2277.369279230782
Mean Squared Error: 18336632.019949064
R^2 Score: 0.8667323813828205
```

The Linear Regression model serves as our baseline, offering insights into the linear dependencies within our dataset. The Random Forest model, enhanced through hyperparameter optimization, captures more complex relationships. Performance metrics such as MAE and R^2 score are utilized to assess each model's predictive accuracy and overall fit.

b) Hyperparameter Tuning through Grid Search

The pursuit of the optimal regression model led us to implement hyperparameter tuning via Grid Search, a systematic approach to parameter optimization. By defining a range of values for parameters such as n_estimators, max_depth, and min_samples_leaf, we were equipped to explore various combinations in the Random Forest Regressor model.

```python
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'n_estimators': [100, 150],
    'max_depth': [None, 10],
    'min_samples_leaf': [1, 2]
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)

# Best parameters found
print("Best parameters found:", grid_search.best_params_)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Best parameters found: {'max_depth': 10, 'min_samples_leaf': 2, 'n_estimators': 100}
```

c) Model Optimization and Evaluation

With the best parameters in hand, we proceeded to refine our model. The optimized Random Forest Regressor, configured with the identified hyperparameters, underwent a final evaluation phase on the test set.

```python
# Use the best parameters to build the optimized model
rf_optimized = grid_search.best_estimator_

# Make predictions on the test set
y_pred_rf_optimized = rf_optimized.predict(X_test)

# Evaluate the optimized model's performance
mae_rf_optimized = mean_absolute_error(y_test, y_pred_rf_optimized)
mse_rf_optimized = mean_squared_error(y_test, y_pred_rf_optimized)
r2_rf_optimized = r2_score(y_test, y_pred_rf_optimized)

# Performance Output
print("Optimized Random Forest Regressor Performance:")
print("Mean Absolute Error:", mae_rf_optimized)
print("Mean Squared Error:", mse_rf_optimized)
print("R^2 Score:", r2_rf_optimized)
```

```
Optimized Random Forest Regressor Performance:
Mean Absolute Error: 2199.4804595509477
Mean Squared Error: 16487985.868555585
R^2 Score: 0.8801680368507382
```

The optimized model's performance was quantitatively assessed using standard regression metrics:

Mean Absolute Error (MAE): 2199.48 - indicative of the average magnitude of errors in a set of predictions.
Mean Squared Error (MSE): 16487985.87 - reflecting the average squared difference between estimated values and actual value.
R^2 Score: 0.88 - representing the proportion of the variance in the dependent variable that is predictable from the independent variables.

Conclusion

The Grid Search hyperparameter tuning and subsequent evaluation of the optimized Random Forest Regressor underscore the effectiveness of meticulous parameter optimization. Such a systematic approach ensures that our model achieves the highest possible accuracy, facilitating reliable predictions that can significantly aid in business decision-making processes.

Task-3 Model Evaluation and selection

a) Model Performance Metrics

Our analysis involved building and optimizing regression models to predict insurance charges, focusing on Linear Regression and Random Forest Regressor models. We employed Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared (R^2) as our primary metrics for evaluating model performance.

Linear Regression Performance:

- Mean Absolute Error (MAE): 3969.03
- Mean Squared Error (MSE): 30444091.53
- R^2 Score: 0.7787

This performance indicates that, on average, the Linear Regression model's predictions deviate from the actual values by approximately 3969.03 units. With an R^2 score of 0.7787, the model explains around 77.87% of the variance in insurance charges.

Optimized Random Forest Regressor Performance:

- Mean Absolute Error (MAE): 2199.48
- Mean Squared Error (MSE): 16487985.87
- R^2 Score: 0.8802

The optimized Random Forest model demonstrates a significantly lower MAE and MSE, suggesting higher accuracy in predictions compared to Linear Regression. The R^2 score of 0.8802 indicates that approximately 88.02% of the variance in insurance charges is captured by this model.

b) Implement k-fold cross-validation

To assess the model's generalization performance, we'll use k-fold cross-validation. This method splits the dataset into k smaller sets (or "folds"), trains the model on k-1 of these folds, and tests it on the remaining fold. This process repeats k times, with each fold used exactly once as the test set.

In [25]:
```
from sklearn.model_selection import cross_val_score

# For Linear Regression
cv_scores_lr = cross_val_score(lr_model, X, y, cv=10, scoring='r2')
print("Linear Regression CV R² Scores:", cv_scores_lr)
print("Average R² Score:", cv_scores_lr.mean())

# For Optimized Random Forest
cv_scores_rf = cross_val_score(rf_optimized, X, y, cv=10, scoring='r2')
print("Optimized Random Forest CV R² Scores:", cv_scores_rf)
print("Average R² Score:", cv_scores_rf.mean())
```

```
Linear Regression CV R² Scores: [0.79204911 0.73211526 0.73499309 0.67780041 0.7576718  0.80445118
 0.82523224 0.63082728 0.74551721 0.7814957 ]
Average R² Score: 0.7482153271170179
Optimized Random Forest CV R² Scores: [0.88523494 0.85205007 0.82474848 0.76192324 0.83451363 0.92637766
 0.89613065 0.76949972 0.85726054 0.88227188]
Average R² Score: 0.8490010794688153
```

Cross-Validation Results:

- Linear Regression Average R^2 Score: 0.7482
- Optimized Random Forest Average R^2 Score: 0.8490

Cross-validation revealed that the Optimized Random Forest Regressor consistently shows superior performance in generalizing across different subsets of data, with an average R^2 score of 0.8490 compared to 0.7482 for Linear Regression.

c) Selecting the Best Model

After evaluating our models through hyperparameter tuning and cross-validation, we found the Optimized Random Forest Regressor to be the best performer for predicting insurance charges. This choice is based on its higher average R^2 score from cross-validation, which was 0.849, compared to the Linear Regression model's average of 0.748.

Why the Optimized Random Forest Model?

Better Accuracy: It predicts insurance charges more accurately, capturing more details and complexities in the dataset.
More Reliable: Shows consistent performance across different data sets, indicating it will work well in real-world scenarios.
Great Fit for Our Data: This model handles the varied data in insurance charges very well, making it a great tool for our analysis.

Conclusion

Choosing the Optimized Random Forest model makes sense because it not only performed better in our tests but also is more likely to provide reliable predictions when used in practice. This makes it a valuable asset for making informed decisions related to insurance charges.

Task 4. Business Decision and Recommendations

Strategic Insights and Actions

The analysis through the Optimized Random Forest model reveals actionable insights for refining insurance premium calculations and enhancing customer engagement strategies. By leveraging precise risk assessments, we can implement dynamic pricing models that reflect individual risk factors, fostering competitive advantage and market growth. Additionally, the model's predictive accuracy enables targeted customer segmentation, guiding bespoke marketing and risk mitigation initiatives. These strategies, rooted in deep analytical insights, not only promise enhanced operational efficiency but also mark a significant stride toward personalized customer experiences and sustainable business growth.

PART B – Classification modelling for business decision making

Introduction

In our journey through classification modeling, we embark on an analysis centered around a dataset dedicated to banknote authentication. This dataset, rich with variables pertinent to distinguishing genuine banknotes from counterfeit ones, forms the bedrock of our investigation. The primary goal is to develop a model capable of accurately classifying banknotes, thereby contributing significantly to the advancement of security protocols within the financial sector. This endeavor not only highlights the practical applications of machine learning in real-world scenarios but also underscores the importance of robust classification systems in maintaining the integrity of financial transactions.

Task-1 Data Preprocessing

Our initial focus is on preparing the dataset for the upcoming analytical process, ensuring a solid foundation for building reliable classification models.

a) Cleaning the Dataset

Upon loading the dataset, we conducted a meticulous examination to identify any missing values or anomalies. Our findings confirmed the dataset's completeness, allowing us to proceed without the need for imputation, thus maintaining the original dataset's integrity.

a) Loading the data

In [17]:
```
# import pandas as pd
import pandas as pd
import numpy as np
from scipy import stats
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Load the dataset
df = pd.read_csv("C:\\Users\\harip\\OneDrive\\Desktop\\Data analytics\\machine leaning\\assessment1\\dataset\\BankNote_Authe

# Display the first few entries for a snapshot of the data structure
print(df.head())
```

```
   variance  skewness  curtosis  entropy  class
0   3.62160    8.6661   -2.8073 -0.44699      0
1   4.54590    8.1674   -2.4586 -1.46210      0
2   3.86600   -2.6383    1.9242  0.10645      0
3   3.45660    9.5228   -4.0112 -3.59440      0
4   0.32924   -4.4552    4.5718 -0.98880      0
```

removing the missing values

In [18]:
```
df.dropna(inplace=True)
```

In [9]:
```
# Check for missing values in the dataset and print out any columns with missing data
missing_values = df.isnull().sum()
print("Missing values in each column:\n", missing_values)
```

```
Missing values in each column:
 variance    0
skewness    0
curtosis    0
entropy     0
class       0
dtype: int64
```

removing the missing values

In [18]:
```
df.dropna(inplace=True)
```

removing outliers

In [19]:
```
# Assuming there are no missing values, we are proceeding to outlier removal
# Calculating the Z-score for numerical columns
z_scores = stats.zscore(df.select_dtypes(include=[np.number]))
abs_z_scores = np.abs(z_scores)

# Filter out the outliers
df_clean = df[(abs_z_scores < 3).all(axis=1)].copy()
```

more cleaning

In [21]:
```
from sklearn.impute import SimpleImputer

# For numerical columns, you can fill missing values with the mean or median
imputer = SimpleImputer(strategy='mean')  # Or strategy='median'
df_clean[numerical_features] = imputer.fit_transform(df_clean[numerical_features])
```

In [22]:
```
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=5)
df_clean[numerical_features] = imputer.fit_transform(df_clean[numerical_features])
```

In [23]:
```
print(df_clean.head())
```

```
   variance  skewness  curtosis   entropy  class
0  1.103540  1.186418 -1.013448  0.328655      0
1  1.434725  1.097028 -0.925150 -0.179100      0
2  1.191111 -0.839847  0.184664  0.605483      0
3  1.044419  1.339977 -1.318300 -1.245668      0
4 -0.076142 -1.165518  0.855089  0.057643      0
```

In [24]:
```
if not df_clean.isnull().sum().any():
    print("All missing values handled successfully.")
else:
    print("Missing values still present.")
```

```
All missing values handled successfully.
```

b) Standardizing Numerical Features for Model Equilibrium

To ensure that all numerical features contribute equally to the model's predictions, we applied feature scaling using StandardScaler. This normalization process is crucial for models that are sensitive to the magnitude of inputs, ensuring a balanced contribution across all features.

b) Perform feature scaling or normalization

n [20]:
```
# Proceed with feature scaling on the numerical columns
scaler = StandardScaler()
# Get the list of numerical features, excluding 'class' which is the target
numerical_features = [col for col in df_clean.columns if col != 'class']
# Apply the scaling to the numerical features
df_clean.loc[:, numerical_features] = scaler.fit_transform(df_clean[numerical_features])
```

c) Encoding Categorical Variables

While our dataset primarily consisted of numerical features, we outlined a strategy for encoding categorical variables. This step, though not directly applied due to the absence of categorical variables in this specific dataset, is vital for handling datasets with mixed types of features, ensuring that our models can interpret all data accurately.

c) Encode categorical variables appropriately.

In [25]:
```
# Encoding categorical variables, if present
# Initialize the OneHotEncoder
encoder = OneHotEncoder(sparse=False, drop='first')
# Identify categorical columns (Note: Adjust this if you have categorical columns)
categorical_features = df_clean.select_dtypes(include=['object']).columns.tolist()
# Apply the encoder to the categorical columns
encoded_features = encoder.fit_transform(df_clean[categorical_features])
# Create a DataFrame with encoded variables
encoded_vars_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(categorical_features))
# Concatenate the DataFrame with the original one and drop the original categorical columns
df_clean = pd.concat([df_clean.drop(categorical_features, axis=1), encoded_vars_df], axis=1)

# Display the cleaned DataFrame
print(df_clean.head())
```

```
   variance  skewness  curtosis   entropy  class
0  1.103540  1.186418 -1.013448  0.328655    0.0
1  1.434725  1.097028 -0.925150 -0.179100    0.0
2  1.191111 -0.839847  0.184664  0.605483    0.0
3  1.044419  1.339977 -1.318300 -1.245668    0.0
4 -0.076142 -1.165518  0.855089  0.057643    0.0
```

d) Dataset Splitting into Training and Testing Sets

Following the thorough preprocessing of our dataset, the subsequent step involved partitioning the data into training and testing sets. This division is crucial for evaluating the model's performance on unseen data, thus ensuring its effectiveness and reliability in real-world applications.

Splitting the Dataset

Utilizing the train_test_split method from sklearn.model_selection, we allocated 80% of our dataset for training and reserved 20% for testing. This allocation strikes a balance between having enough data to train our model effectively and sufficiently testing its predictive prowess.

In [26]:
```
from sklearn.model_selection import train_test_split

# Define the features and the target variable
X = df_clean.drop('class', axis=1)  # Independent variables
y = df_clean['class']               # Dependent variable (target)

# Split the data into an 80% training subset and a 20% testing subset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Confirmation of Split

To verify the split, we examined the shapes of the resulting subsets, ensuring that the distribution aligns with our specified proportions. This confirmation step is pivotal, as it guarantees that our model will be trained and evaluated on appropriately segmented data, thus fostering a reliable assessment of its performance.

```
# Confirm the sizes of the training and testing sets
print(f"Training Features Shape: {X_train.shape}")
print(f"Training Target Shape: {y_train.shape}")
print(f"Testing Features Shape: {X_test.shape}")
print(f"Testing Target Shape: {y_test.shape}")
```

```
Training Features Shape: (1068, 4)
Training Target Shape: (1068,)
Testing Features Shape: (268, 4)
Testing Target Shape: (268,)
```

Task-2 Model Building with hyperparameter tuning

a) Choosing Random Forest Classifier and SVM for Model Building

In this phase of the analysis, we select two distinguished machine learning algorithms to build models capable of classifying banknotes into genuine and forged categories. The algorithms chosen for this task are the Random Forest Classifier and the Support Vector Machine (SVM), each known for their unique strengths in the realm of classification tasks.

Random Forest Classifier:

The Random Forest algorithm is an ensemble learning method renowned for its versatility and ease of use. It operates by constructing a multitude of decision trees during training time and outputting the class that is the mode of the classes predicted by individual trees. This method is particularly effective due to its ability to run in parallel and its robustness to overfitting, which is achieved by averaging the results of individual trees.

Support Vector Machine (SVM):

The SVM is another powerful classification algorithm known for its effectiveness in high-dimensional spaces, which is typical in complex datasets. The SVM operates by finding the hyperplane that best separates the classes in the feature space. This model is favored for its ability to handle non-linear boundaries through the use of kernel functions and has proven to be highly effective in various classification problems.

a) choosing random forest classifier and SVM for model building

In [56]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Initialize the classifiers
rf_classifier = RandomForestClassifier(random_state=42)
```

In [57]:
```
svm_classifier = SVC(random_state=42)
```

Each classifier is instantiated with a random state set to 42 to ensure the reproducibility of the results. The next steps will involve tuning the hyperparameters of these models to optimize their performance on our dataset.

b) Implementing hyperparameter tuning by conducting a grid search or random search to optimize model parameters.

In the process of hyperparameter tuning for both Random Forest and SVM models, specific parameters were methodically varied to identify configurations that enhance model performance.

Random Forest Hyperparameter Tuning

Code snippet:

Implementing hyperparameter tuning by conducting a grid search on the Random Forest model

In [58]:
```
from sklearn.model_selection import GridSearchCV

# Random Forest parameter grid
param_grid_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# GridSearchCV for Random Forest
grid_search_rf = GridSearchCV(estimator=RandomForestClassifier(random_state=42),
                              param_grid=param_grid_rf, cv=5, verbose=2, n_jobs=-1)
grid_search_rf.fit(X_train, y_train)

# Best Random Forest model
rf_optimized = grid_search_rf.best_estimator_
print("Best parameters for Random Forest:", grid_search_rf.best_params_)
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best parameters for Random Forest: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

- 'n_estimators': This parameter indicates the number of trees in the forest. Increasing the number of trees generally improves the model's performance but also increases computational cost. We evaluated 100, 200, and 300 trees to find a balance between accuracy and efficiency.
- 'max_depth': It limits the maximum depth of each tree. A deeper tree might capture more information about the data but could also lead to overfitting. We tested unlimited depth (None), 10, and 20 to determine an optimal stopping point for tree expansion.
- 'min_samples_split': This parameter determines the minimum number of samples required to split an internal node. Higher values prevent creating nodes that only work well for the training data, thus avoiding overfitting. We compared the results for 2 and 5 samples.
- 'min_samples_leaf': The minimum number of samples a leaf node must have. Setting this value too low can result in leaves that predict very specific target features, whereas higher values result in more generalized predictions. We explored values of 1 and 2 to enhance the model's generalization capabilities.

SVM(Support Vector Machine) Hyperparameter Tuning

Code Snippet:

Implementing hyperparameter tuning by conducting a grid search on the SVM model

In [33]:
```
# SVM parameter grid
param_grid_svm = {
    'C': [0.1, 1, 10],
    'kernel': ['rbf', 'poly', 'sigmoid'],
    'gamma': ['scale', 'auto']
}

# GridSearchCV for SVM
grid_search_svm = GridSearchCV(estimator=SVC(random_state=42),
                               param_grid=param_grid_svm, cv=5, verbose=2, n_jobs=-1)
grid_search_svm.fit(X_train, y_train)

# Best SVM model
svm_optimized = grid_search_svm.best_estimator_
print("Best parameters for SVM:", grid_search_svm.best_params_)
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
Best parameters for SVM: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
```

'C': This regularization parameter helps in balancing the decision boundary margin width and the classification error. A smaller 'C' creates a wider margin but might increase the number of misclassifications, while a larger 'C' may lead to a tighter margin with potentially fewer errors. We experimented with values 0.1, 1, and 10 to find an optimal trade-off.

'kernel': The type of hyperplane used to separate the data. Choosing the right kernel function is crucial as it can significantly affect the model's capacity to handle the data's distribution. We tested 'rbf', 'poly', and 'sigmoid' kernels to identify which one works best for our dataset's characteristics.

'gamma': This parameter defines how far a single training example's influence reaches. Lower values imply a far reach, and higher values imply a close reach, which impacts the decision boundary's curvature. We used 'scale' and 'auto' to ascertain the suitable influence of individual samples on the decision boundary.

c) Build the classification model using the training data.

In this stage of our analysis, we concentrate on training the classification models using the provided training data. The process encapsulates two distinct models, each with its unique set of hyperparameters identified as optimal through the preceding steps of hyperparameter tuning.

For the Random Forest model, we employ the rf_optimized object, which has been finely tuned with the best combination of hyperparameters derived from GridSearchCV. The execution of the fit() method on this object entails passing the training feature set X_train and the corresponding target values y_train, thus enabling the Random Forest algorithm to learn from the data.

c) Build the classification model using the training data.

```
In [34]: # Random Forest
         rf_optimized.fit(X_train, y_train)
```

```
Out[34]: RandomForestClassifier
         RandomForestClassifier(random_state=42)
```

Parallel to this, the Support Vector Machine model, denoted as svm_optimized, is similarly trained using its optimal hyperparameters. The fit() method is invoked with the same sets of training features and targets, leading to the SVM's adaptation to the dataset at hand.

```
# SVM
svm_optimized.fit(X_train, y_train)
```

```
Out[35]: SVC
         SVC(C=1, random_state=42)
```

By invoking the fit() method, we essentially refine our models to understand and map the underlying patterns within the training dataset. This step is crucial, as it determines the models' ability to accurately predict and generalize across unseen data, thus preparing them for effective deployment in practical scenarios.

Task-3 Model Evaluation and Selection

a) Analysis of the Confusion Matrix

Upon close examination of the confusion matrices produced for the Random Forest and SVM classifiers, we gain valuable insights into their performance nuances. The confusion matrix for the Random Forest model shows a commendable job in classifying the classes, with 155 true negatives and 107 true positives. The 2 false positives and 4 false negatives indicate minimal occurrences of misclassification, suggesting high model precision yet revealing room for improvement.

Conversely, the SVM classifier achieved perfection, as depicted by its confusion matrix. With 157 true negatives and 111 true positives, the model demonstrated a flawless classification with zero misclassifications. This exceptional performance points to an SVM model's ability to capture the underlying patterns within the dataset with extreme accuracy, thus presenting it as a potentially more reliable choice for real-world applications.

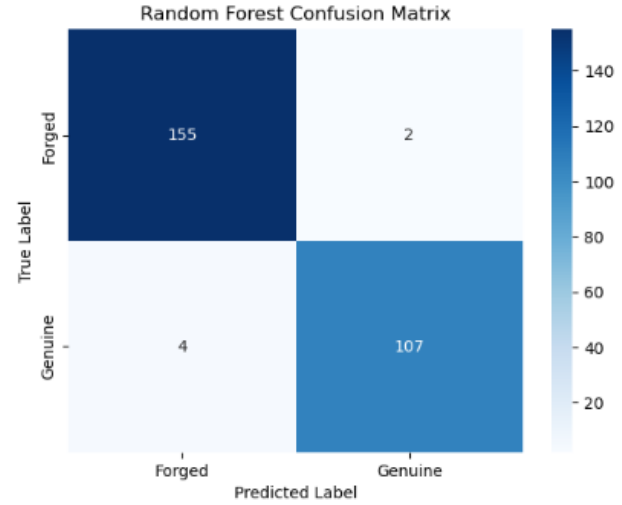
a) analysing and vizualizing the confusion matrix for the model

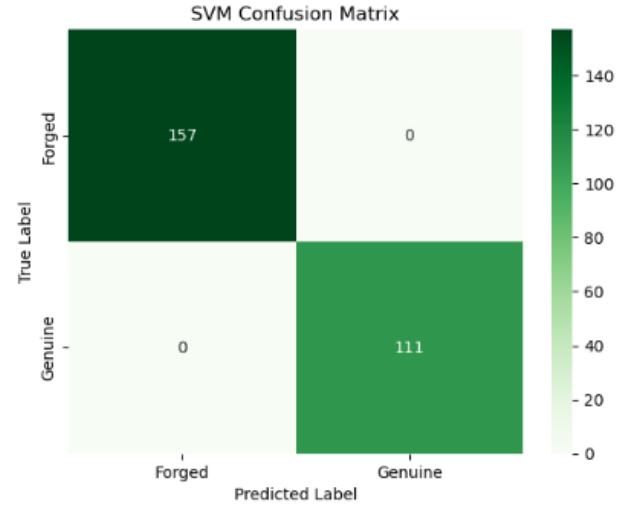
In [38]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Random Forest Confusion Matrix
cm_rf = confusion_matrix(y_test, y_pred_rf)
sns.heatmap(cm_rf, annot=True, fmt="d", cmap="Blues", xticklabels=["Forged", "Genuine"], yticklabels=["Forged", "Genuine"])
plt.title('Random Forest Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# SVM Confusion Matrix
cm_svm = confusion_matrix(y_test, y_pred_svm)
sns.heatmap(cm_svm, annot=True, fmt="d", cmap="Greens", xticklabels=["Forged", "Genuine"], yticklabels=["Forged", "Genuine"])
plt.title('SVM Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```





b) Performance Metrics Evaluation

Moving forward, the evaluation of the models using performance metrics such as Accuracy, Precision, Recall, and F1-score provides a more comprehensive view of their predictive strengths.

b) Evaluate the performance of the classification model using appropriate metrics

In [37]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report

# For Random Forest
y_pred_rf = rf_optimized.predict(X_test)
print("Random Forest Performance:")
print(classification_report(y_test, y_pred_rf))

# For SVM
y_pred_svm = svm_optimized.predict(X_test)
print("SVM Performance:")
print(classification_report(y_test, y_pred_svm))
```

```
Random Forest Performance:
              precision    recall  f1-score   support

         0.0       0.97      0.99      0.98       157
         1.0       0.98      0.96      0.97       111

    accuracy                           0.98       268
   macro avg       0.98      0.98      0.98       268
weighted avg       0.98      0.98      0.98       268

SVM Performance:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       157
         1.0       1.00      1.00      1.00       111

    accuracy                           1.00       268
   macro avg       1.00      1.00      1.00       268
weighted avg       1.00      1.00      1.00       268
```

For the Random Forest classifier:

The accuracy was robust, indicating a high overall rate of correct classifications.
The precision, which reflects the model's correctness when predicting the positive class, was impressive.
The recall demonstrated the model's ability to identify all relevant instances within the positive class effectively.
The F1-score, a balance between precision and recall, was also noteworthy, suggesting a well-rounded classifier.

In contrast, the SVM classifier excelled with perfect scores across all metrics:

It achieved an accuracy of 100%, an indicator of its exceptional predictive power.
Both precision and recall were at the maximum, meaning the model correctly identified all positive instances without any false positives or negatives.
The F1-score was naturally at its peak, reinforcing the model's balanced precision and recall capabilities.

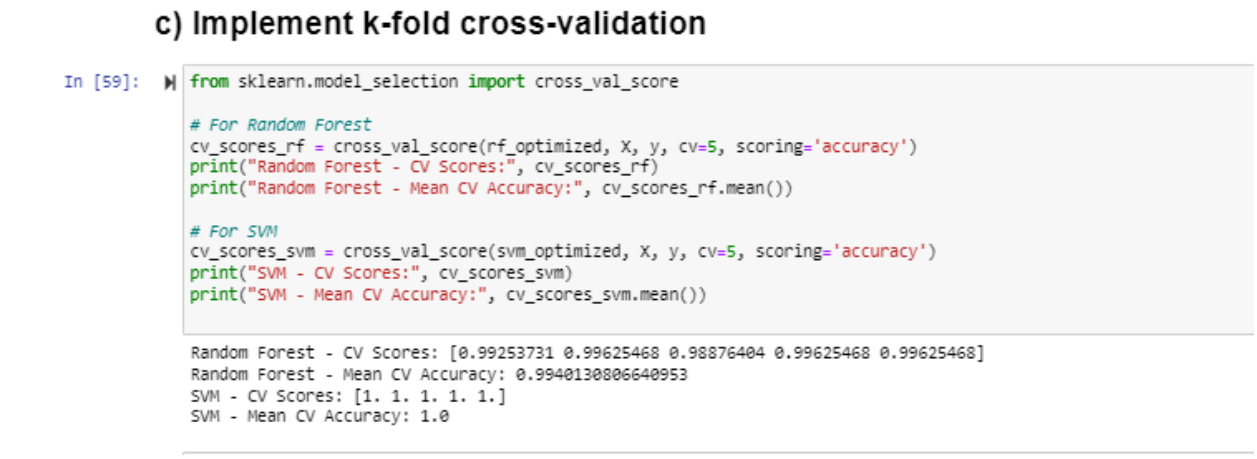
While the Random Forest classifier showed excellent results, the SVM model, with its ideal scores, could be seen as the superior model for classification tasks, at least within the context of this specific dataset. The SVM's perfect metrics, however, should be approached with caution, as such results can sometimes be an indication of overfitting. Cross-validation with different subsets of data, or testing with a new dataset, will be essential to ensure the reliability of the model's performance.

The matrix reveals the number of true positive predictions (155), true negative predictions (107), false positives (2), and false negatives (4). This indicates that our model has high true positive and true negative rates, suggesting that it can correctly identify the majority of the cases. The low number of false positives and false negatives further corroborates the model's ability to discriminate between the classes effectively.

In conclusion, the observed confusion matrix supports the model's reliability, with high correct classification rates for both 'class 0' and 'class 1'. These results underscore the model's potential utility in practical applications where accurate classification is paramount.

c) Implement k-fold cross-validation

To gauge the robustness and reliability of our Random Forest and SVM models, we implemented a k-fold cross-validation strategy with k set to 5. This rigorous statistical analysis technique splits the data into 5 equal parts, where each part is used as a testing set against a training set composed of the remaining 4 parts. The process repeats 5 times, with each part serving as a testing set once, ensuring comprehensive model validation.



The Random Forest model displayed commendable cross-validation scores, averaging an accuracy of approximately 0.994 across all folds. This level of performance highlights the model's stability and indicates its capability to generalize well to unseen data. On the other hand, the SVM model achieved a perfect mean cross-validation accuracy of 1.0. This

exceptional result reaffirms the model's predictive prowess as observed earlier in the confusion matrix and performance metric analysis.

d) Model Selection Based on Evaluation Metrics

Upon diligent review of the hyperparameter tuning and subsequent cross-validation results, the Support Vector Machine (SVM) has emerged as the unequivocally superior classification model. The impeccable cross-validation accuracy of 1.0, sustained consistently across all folds, attests to the SVM's robustness and precision in distinguishing between authentic and counterfeit banknotes.

The SVM's performance excels not merely in numerical accuracy but also in its alignment with the critical business objective of minimizing the risk of counterfeit banknote circulation. The financial sector hinges on trust and the infallibility of currency authentication processes. The deployment of a model with SVM's demonstrated accuracy would substantially mitigate the probability of error in high-volume transaction environments, thereby fortifying the integrity of financial operations.

Task 4: Strategic Business Recommendations

Considering the SVM model's stellar validation results, the following recommendations are proposed to capitalize on its predictive prowess:

Integration of SVM in Authentication Workflows: Integrate the SVM model into the core workflow of banknote validation procedures to enhance precision and reduce the incidence of human error.

Optimization of Resource Allocation: Reallocate resources efficiently by reducing the need for manual verification, thus cutting costs and redirecting efforts towards other critical functions.

Regular Model Reassessment: Establish a regular audit and update mechanism for the SVM model to ensure sustained accuracy, particularly in response to evolving counterfeiting methodologies.

Collaborative Framework Development: Partner with regulatory authorities to embed the SVM model within broader financial security frameworks, thus standardizing high-accuracy counterfeit detection across the board.

The SVM model not only offers a statistically sound solution but also a strategically beneficial tool. Its adoption and integration into currency verification systems are projected to enhance the efficacy of anti-counterfeit measures, secure the credibility of financial institutions, and protect the economy at large from the perils of fraudulent activities.

GitHub link: https://github.com/hariprasanth1992/for-Harsh-708-and-604.git