SEPTEMBER 21, 2023

Customer churn prediction

Documentation

Done by:

1- Mohammad Riyad Al Smadi
2- Fatimah Shareef Alzaareer
3- Mahmoud Kouf

Orange Coding Academy

Table of Contents

1. Introduction

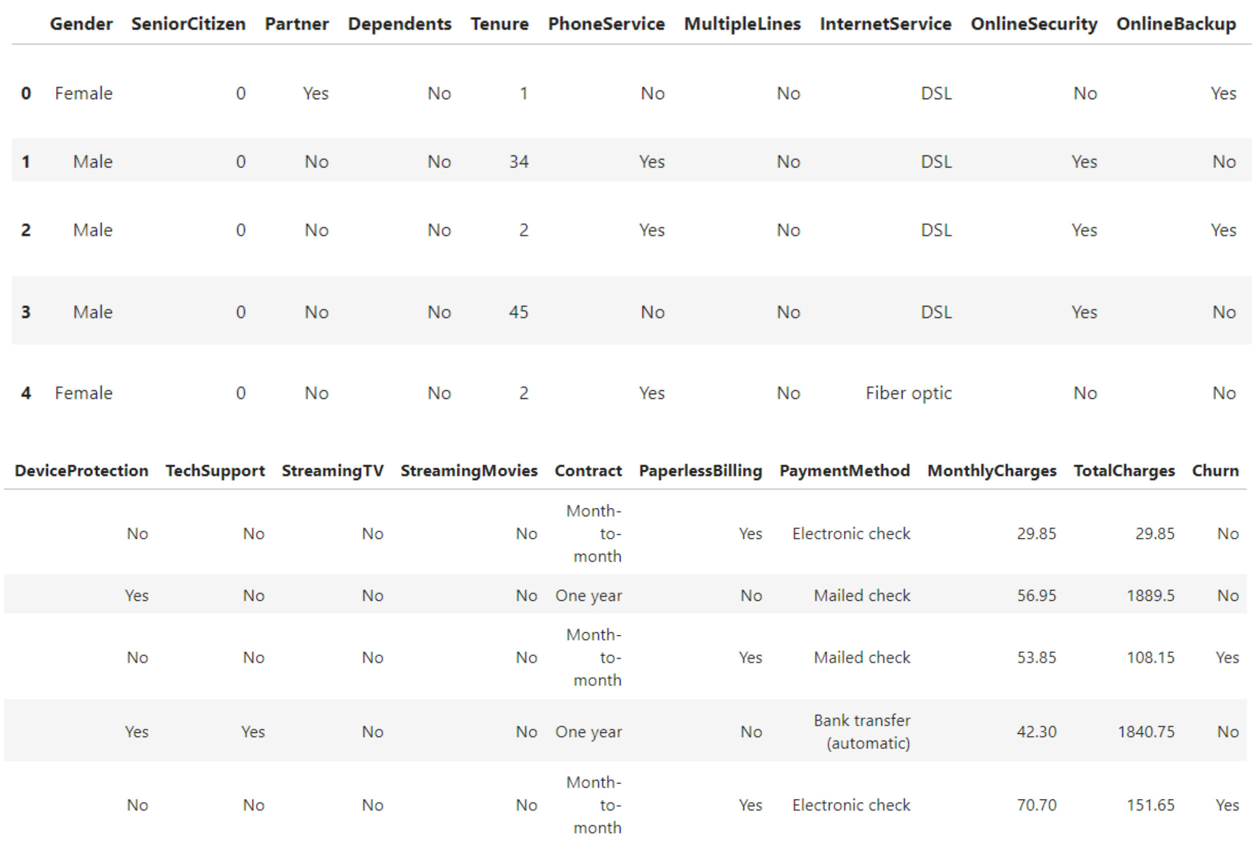
Churn, the loss of customers or users, can have a significant impact on businesses and organizations. Accurately predicting and mitigating churn is a critical task for customer retention and revenue growth. This documentation is designed to provide a comprehensive guide on how to understand, use, and deploy our machine learning model for churn prediction.

In today's competitive landscape, retaining customers is as important as acquiring new ones. Customer churn not only leads to revenue loss but can also indicate underlying issues within a business, such as poor product satisfaction or ineffective marketing strategies. By identifying potential churners in advance, businesses can take proactive measures to retain customers and improve their overall experience.

Our churn prediction model is built on state-of-the-art machine learning techniques, designed to analyze historical data and forecast which customers are most likely to churn in the future. It can be a powerful tool for businesses across various industries, including telecommunications, e-commerce, subscription-based services, and more. By leveraging this model, you can make data-driven decisions to reduce churn, optimize customer engagement, and ultimately boost your bottom line.

This documentation is structured to provide a step-by-step guide on how to work with our churn prediction model. The documentation focuses majorly on model development and deployment.

2. The dataset



	Gender	SeniorCitizen	Partner	Dependents	Tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup
0	Female	0	Yes	No	1	No	No	DSL	No	Yes
1	Male	0	No	No	34	Yes	No	DSL	Yes	No
2	Male	0	No	No	2	Yes	No	DSL	Yes	Yes
3	Male	0	No	No	45	No	No	DSL	Yes	No
4	Female	0	No	No	2	Yes	No	Fiber optic	No	No

DeviceProtection	TechSupport	StreamingTV	StreamingMovies	Contract	PaperlessBilling	PaymentMethod	MonthlyCharges	TotalCharges	Churn
No	No	No	No	Month-to-month	Yes	Electronic check	29.85	29.85	No
Yes	No	No	No	One year	No	Mailed check	56.95	1889.5	No
No	No	No	No	Month-to-month	Yes	Mailed check	53.85	108.15	Yes
Yes	Yes	No	No	One year	No	Bank transfer (automatic)	42.30	1840.75	No
No	No	No	No	Month-to-month	Yes	Electronic check	70.70	151.65	Yes

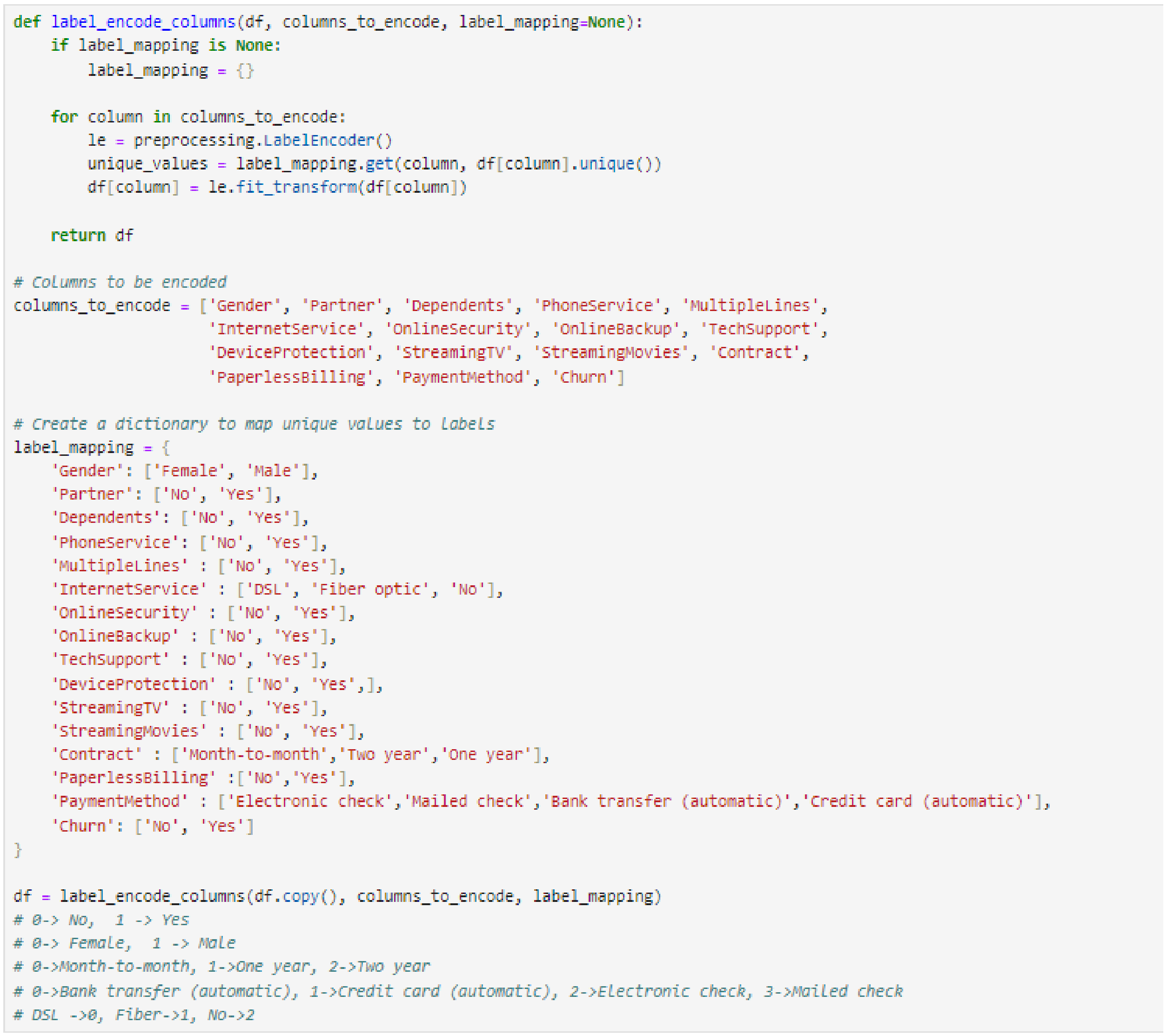
Fig 2.1: The dataset

The dataset includes information about:

- Customers who left within the last month – the column is called Churn.
- Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies.
- Customer account information – how long they had been a customer, contract, payment method, paperless billing, monthly charges, and total charges.
- Demographic info about customers – gender, age range, and if they have partners and dependents.

3. Label Encoding for Categorical Features

This code defines a Python function named label_encode_columns, which is used to label encode categorical columns in a DataFrame. Label encoding is a technique where categorical values are replaced with unique integer labels. Below is a line-by-line explanation of the code:



```python
def label_encode_columns(df, columns_to_encode, label_mapping=None):
    if label_mapping is None:
        label_mapping = {}

    for column in columns_to_encode:
        le = preprocessing.LabelEncoder()
        unique_values = label_mapping.get(column, df[column].unique())
        df[column] = le.fit_transform(df[column])

    return df

# Columns to be encoded
columns_to_encode = ['Gender', 'Partner', 'Dependents', 'PhoneService', 'MultipleLines',
                     'InternetService', 'OnlineSecurity', 'OnlineBackup', 'TechSupport',
                     'DeviceProtection', 'StreamingTV', 'StreamingMovies', 'Contract',
                     'PaperlessBilling', 'PaymentMethod', 'Churn']

# Create a dictionary to map unique values to labels
label_mapping = {
    'Gender': ['Female', 'Male'],
    'Partner': ['No', 'Yes'],
    'Dependents': ['No', 'Yes'],
    'PhoneService': ['No', 'Yes'],
    'MultipleLines' : ['No', 'Yes'],
    'InternetService' : ['DSL', 'Fiber optic', 'No'],
    'OnlineSecurity' : ['No', 'Yes'],
    'OnlineBackup' : ['No', 'Yes'],
    'TechSupport' : ['No', 'Yes'],
    'DeviceProtection' : ['No', 'Yes',],
    'StreamingTV' : ['No', 'Yes'],
    'StreamingMovies' : ['No', 'Yes'],
    'Contract' : ['Month-to-month','Two year','One year'],
    'PaperlessBilling' :['No','Yes'],
    'PaymentMethod' : ['Electronic check','Mailed check','Bank transfer (automatic)','Credit card (automatic)'],
    'Churn': ['No', 'Yes']
}

df = label_encode_columns(df.copy(), columns_to_encode, label_mapping)
# 0-> No,  1 -> Yes
# 0-> Female,  1 -> Male
# 0->Month-to-month, 1->One year, 2->Two year
# 0->Bank transfer (automatic), 1->Credit card (automatic), 2->Electronic check, 3->Mailed check
# DSL ->0, Fiber->1, No->2
```

Fig 3.1: Label encoding code

1 - Importing preprocessing module from scikit-learn library.

```
from sklearn import preprocessing
```

2 - This defines a function called label_encode_columns that takes three arguments:

- df: The input DataFrame that you want to encode.
- columns_to_encode: A list of column names that you want to label encode.
- label_mapping: An optional dictionary that maps column names to lists of unique values. If not provided, an empty dictionary is created.

```
def label_encode_columns(df, columns_to_encode,
label_mapping=None):
```

3 - This checks if label_mapping is provided. If not, it initializes an empty dictionary.

```
if label_mapping is None:

        label_mapping = {}
```

4 - This starts a loop over the columns_to_encode list, where each column is a categorical column that needs encoding.

It creates a LabelEncoder object called le from the preprocessing module of scikit-learn library.

```
for column in columns_to_encode:

     le = preprocessing.LabelEncoder()
```

5 - This line retrieves the unique values associated with the current column from the label_mapping dictionary if available. If not found, it uses the unique values found in the DataFrame's column using df[column].unique().

```
unique_values = label_mapping.get(column, df[column].unique())
```

6 - The fit_transform method of the LabelEncoder object is used to transform the values in the DataFrame's column into integer labels.

The original column in the DataFrame is replaced with the encoded values.

```
df[column] = le.fit_transform(df[column])
```

7 - This line makes a copy of the original DataFrame df and then calls the label_encode_columns function with the copy.

It encodes the columns specified in columns_to_encode using the provided label_mapping or the default mapping.

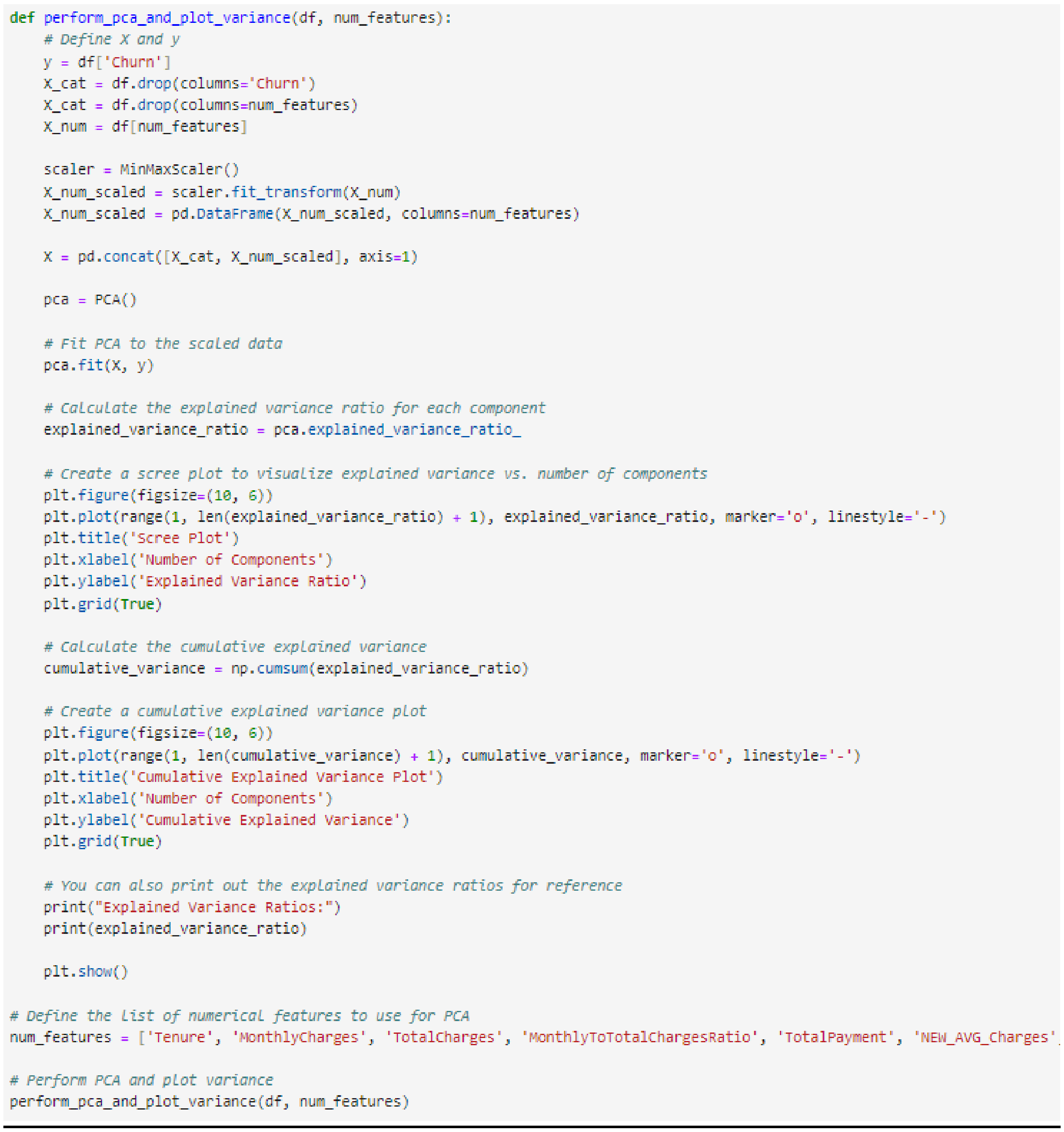
The encoded DataFrame is assigned back to the variable df.

```
df = label_encode_columns(df.copy(), columns_to_encode,
label_mapping)
```

```
# 0-> No,  1 -> Yes

# 0-> Female,  1 -> Male

# 0->Month-to-month, 1->One year, 2->Two year

# 0->Bank transfer (automatic), 1->Credit card (automatic), 2->Electronic check, 3-
>Mailed check

# DSL ->0, Fiber->1, No->2
```

4. Principal Component Analysis (PCA)

This code defines a Python function named perform_pca_and_plot_variance that performs Principal Component Analysis (PCA) on a dataset and plots the explained variance to help determine the optimal number of components to retain.



```python
def perform_pca_and_plot_variance(df, num_features):
    # Define X and y
    y = df['Churn']
    X_cat = df.drop(columns='Churn')
    X_cat = df.drop(columns=num_features)
    X_num = df[num_features]

    scaler = MinMaxScaler()
    X_num_scaled = scaler.fit_transform(X_num)
    X_num_scaled = pd.DataFrame(X_num_scaled, columns=num_features)

    X = pd.concat([X_cat, X_num_scaled], axis=1)

    pca = PCA()

    # Fit PCA to the scaled data
    pca.fit(X, y)

    # Calculate the explained variance ratio for each component
    explained_variance_ratio = pca.explained_variance_ratio_

    # Create a scree plot to visualize explained variance vs. number of components
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, marker='o', linestyle='-')
    plt.title('Scree Plot')
    plt.xlabel('Number of Components')
    plt.ylabel('Explained Variance Ratio')
    plt.grid(True)

    # Calculate the cumulative explained variance
    cumulative_variance = np.cumsum(explained_variance_ratio)

    # Create a cumulative explained variance plot
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='-')
    plt.title('Cumulative Explained Variance Plot')
    plt.xlabel('Number of Components')
    plt.ylabel('Cumulative Explained Variance')
    plt.grid(True)

    # You can also print out the explained variance ratios for reference
    print("Explained Variance Ratios:")
    print(explained_variance_ratio)

    plt.show()

# Define the list of numerical features to use for PCA
num_features = ['Tenure', 'MonthlyCharges', 'TotalCharges', 'MonthlyToTotalChargesRatio', 'TotalPayment', 'NEW_AVG_Charges'

# Perform PCA and plot variance
perform_pca_and_plot_variance(df, num_features)
```

Fig 4.1: PCA code

1 - This defines a function called perform_pca_and_plot_variance that takes two arguments:

- df: The input DataFrame containing the data to be analyzed.
- num_features: A list of column names representing numerical features to be used in PCA.

```
def perform_pca_and_plot_variance(df, num_features):
```

2 - The first two lines create a copy of the DataFrame without the Churn column and the numerical features to only keep categorical features in it.

The last line creates a DataFrame X_num containing only the numerical features specified in num_features.

```
X_cat = df.drop(columns='Churn')

X_cat = df.drop(columns=num_features)

X_num = df[num_features]
```

3 - These lines standardize the numerical features using MinMaxScaler from scikit-learn library. Standardization is common in PCA to ensure that features have similar scales. The scaled features are then stored in X_num_scaled.

```
scaler = MinMaxScaler()

X_num_scaled = scaler.fit_transform(X_num)

X_num_scaled = pd.DataFrame(X_num_scaled, columns=num_features)
```

4 - This line concatenates the scaled numerical features (X_num_scaled) with the categorical features (X_cat) along the columns (axis=1). The result is a single DataFrame X containing all the features.

```
X = pd.concat([X_cat, X_num_scaled], axis=1)
```

5 - PCA is fit to the data (X) with the target variable (y). This step computes the principal components and their explained variances.

The explained_variance_ratio variable now holds the proportion of the total variance explained by each principal component.

```
pca = PCA()

pca.fit(X, y)

explained_variance_ratio = pca.explained_variance_ratio_
```

6 - This block of code creates a scree plot, a graphical representation of the explained variance ratio for each principal component. It helps determine how many components should be retained for dimensionality reduction.

```
plt.figure(figsize=(10, 6))

plt.plot(range(1, len(explained_variance_ratio) + 1),
explained_variance_ratio, marker='o', linestyle='-')

plt.title('Scree Plot')

plt.xlabel('Number of Components')

plt.ylabel('Explained Variance Ratio')

plt.grid(True)
```

7 - cumulative_variance now contains the cumulative explained variance for each principal component. This helps you understand how much total variance is explained as you include more components.

```
cumulative_variance = np.cumsum(explained_variance_ratio)
```

8 - This block of code creates a plot showing the cumulative explained variance as you add more principal components. It helps you decide how many components to retain to capture a desired amount of variance.

```
plt.figure(figsize=(10, 6))

plt.plot(range(1, len(cumulative_variance) + 1),
cumulative_variance, marker='o', linestyle='-')

plt.title('Cumulative Explained Variance Plot')
plt.xlabel('Number of Components')

plt.ylabel('Cumulative Explained Variance')

plt.grid(True)
```

Finally, this line displays the created plots.

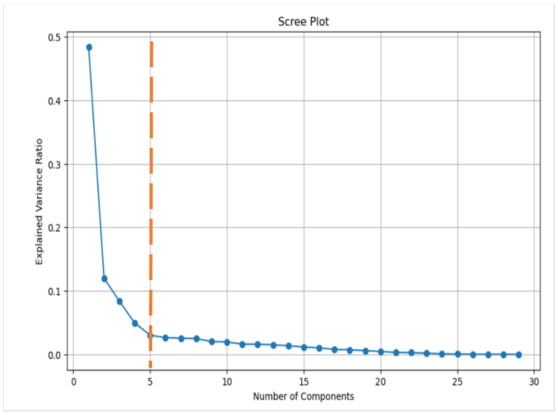


Fig 4.2: Scree Plot

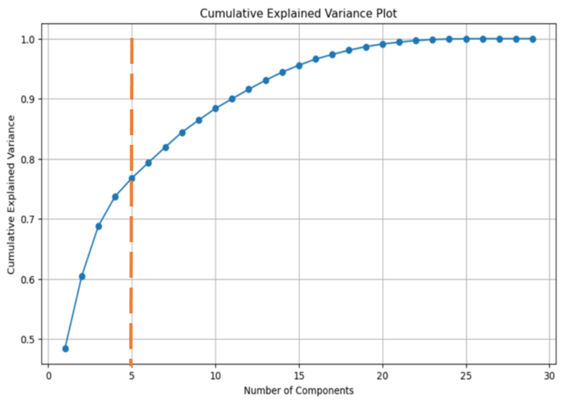


Fig 4.3: Cumulative Variance Plot

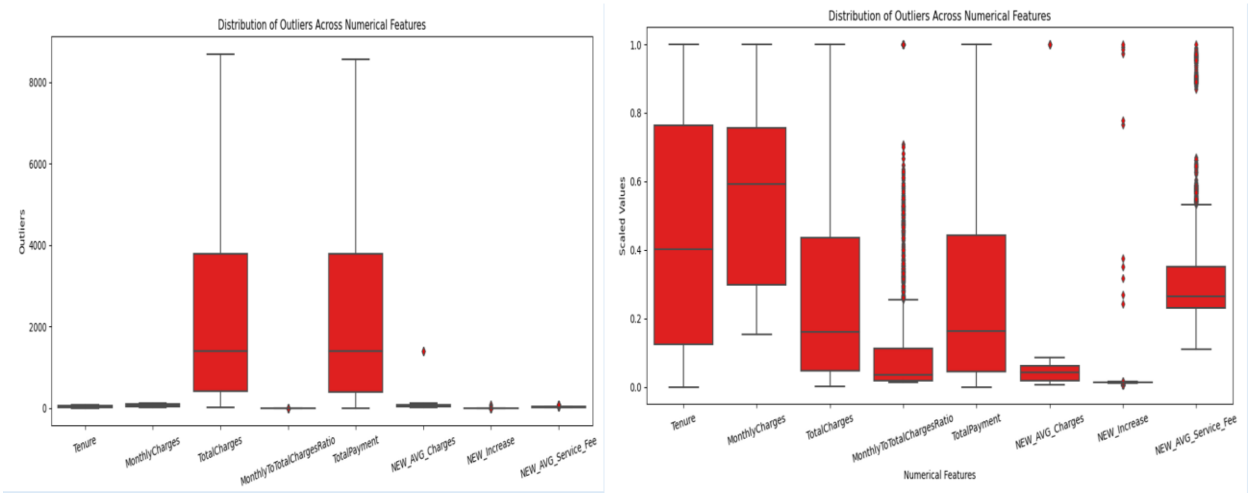
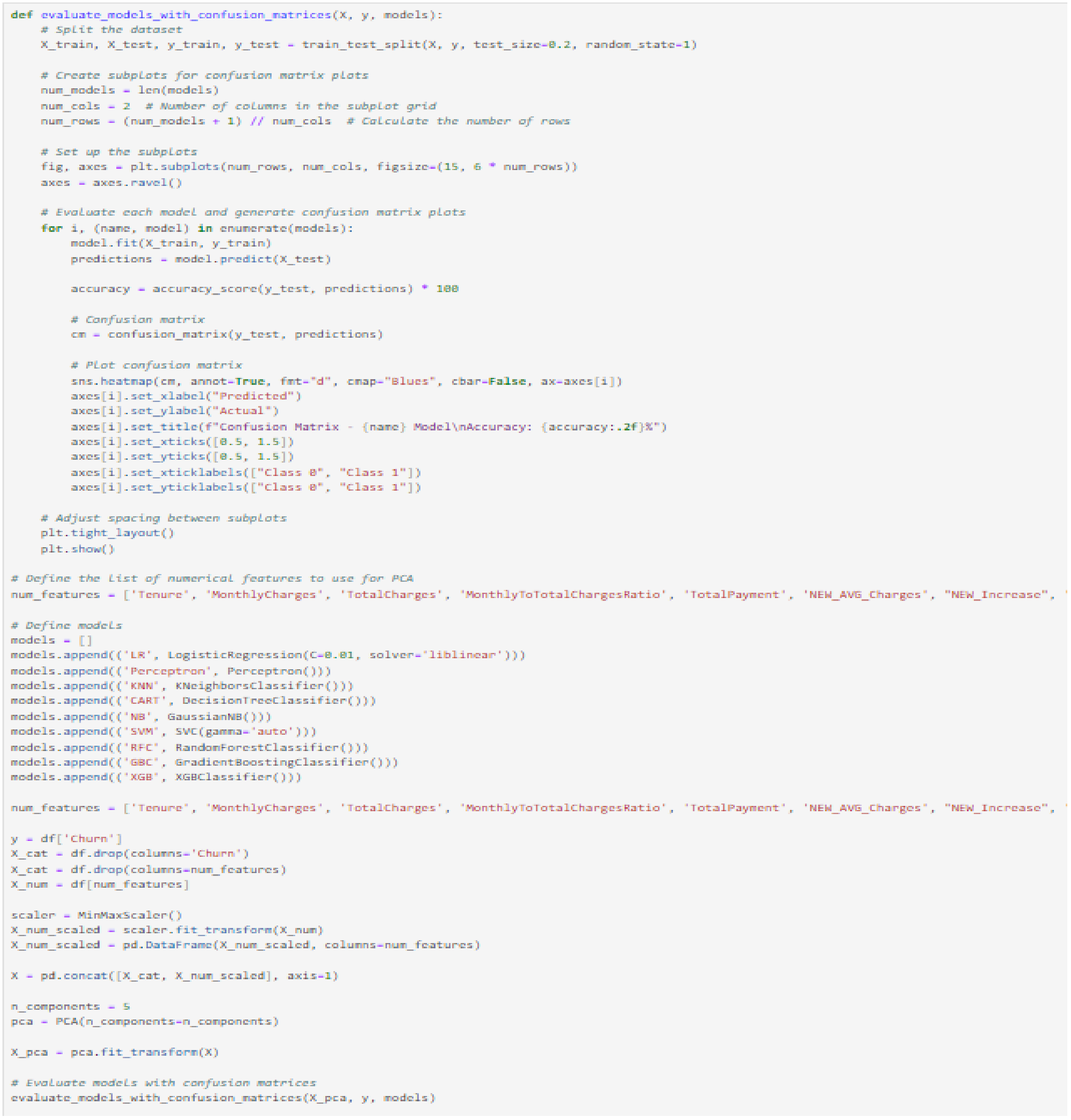


Fig 4.4: Data before and after normalization

5. Comparing Classification Models



```python
def evaluate_models_with_confusion_matrices(X, y, models):
    # Split the dataset
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

    # Create subplots for confusion matrix plots
    num_models = len(models)
    num_cols = 2  # Number of columns in the subplot grid
    num_rows = (num_models + 1) // num_cols  # Calculate the number of rows

    # Set up the subplots
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 6 * num_rows))
    axes = axes.ravel()

    # Evaluate each model and generate confusion matrix plots
    for i, (name, model) in enumerate(models):
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)

        accuracy = accuracy_score(y_test, predictions) * 100

        # Confusion matrix
        cm = confusion_matrix(y_test, predictions)

        # Plot confusion matrix
        sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False, ax=axes[i])
        axes[i].set_xlabel("Predicted")
        axes[i].set_ylabel("Actual")
        axes[i].set_title(f"Confusion Matrix - {name} Model\nAccuracy: {accuracy:.2f}%")
        axes[i].set_xticks([0.5, 1.5])
        axes[i].set_yticks([0.5, 1.5])
        axes[i].set_xticklabels(["Class 0", "Class 1"])
        axes[i].set_yticklabels(["Class 0", "Class 1"])

    # Adjust spacing between subplots
    plt.tight_layout()
    plt.show()

# Define the list of numerical features to use for PCA
num_features = ['Tenure', 'MonthlyCharges', 'TotalCharges', 'MonthlyToTotalChargesRatio', 'TotalPayment', 'NEW_AVG_Charges', "NEW_Increase", '

# Define models
models = []
models.append(('LR', LogisticRegression(C=0.01, solver='liblinear')))
models.append(('Perceptron', Perceptron()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
models.append(('RFC', RandomForestClassifier()))
models.append(('GBC', GradientBoostingClassifier()))
models.append(('XGB', XGBClassifier()))

num_features = ['Tenure', 'MonthlyCharges', 'TotalCharges', 'MonthlyToTotalChargesRatio', 'TotalPayment', 'NEW_AVG_Charges', "NEW_Increase", '

y = df['Churn']
X_cat = df.drop(columns='Churn')
X_cat = df.drop(columns=num_features)
X_num = df[num_features]

scaler = MinMaxScaler()
X_num_scaled = scaler.fit_transform(X_num)
X_num_scaled = pd.DataFrame(X_num_scaled, columns=num_features)

X = pd.concat([X_cat, X_num_scaled], axis=1)

n_components = 5
pca = PCA(n_components=n_components)

X_pca = pca.fit_transform(X)

# Evaluate models with confusion matrices
evaluate_models_with_confusion_matrices(X_pca, y, models)
```

Fig 5.1: Comparing models code

This code defines a function evaluate_models_with_confusion_matrices that evaluates a list of machine learning models by generating confusion matrices for each model's predictions.

1 - This function is defined to evaluate multiple machine learning models using confusion matrices.

It takes three arguments:

- X: The feature matrix (input data).
- y: The target variable (labels).
- models: A list of tuples, where each tuple contains the model name and the model object.

```
def evaluate_models_with_confusion_matrices(X, y, models):
```

2 - The input data is split into training and testing sets using train_test_split from scikit-learn. It's common to use an 80/20 split (80% training, 20% testing).

```
X_train, X_test, y_train, y_test = train_test_split(X, y, 
test_size=0.2, random_state=1)
```

3 - This code sets up subplots for displaying confusion matrices. The number of rows and columns for the subplot grid is determined based on the number of models to be evaluated.

```
num_models = len(models)

num_cols = 2  # Number of columns in the subplot grid

num_rows = (num_models + 1) // num_cols  # Calculate the number 
of rows



fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 6 * 
num_rows))

axes = axes.ravel()
```

4 - This loop iterates through the list of models and performs the following steps for each model:

- Fit the model to the training data (X_train and y_train).
- Make predictions on the test data (X_test) using the trained model.
- Calculate the accuracy of the model's predictions.

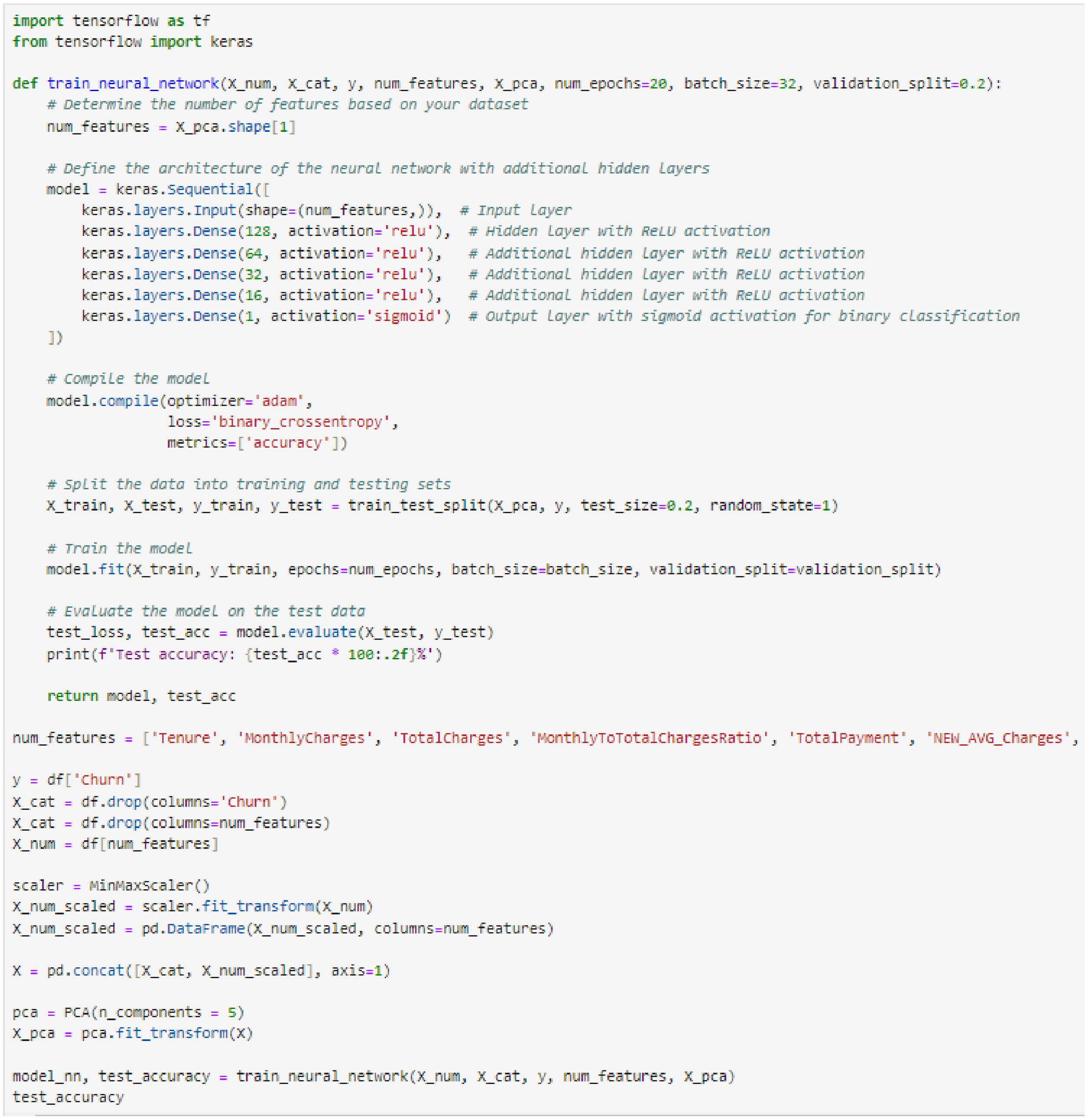
```
for i, (name, model) in enumerate(models):

        model.fit(X_train, y_train)

        predictions = model.predict(X_test)

        accuracy = accuracy_score(y_test, predictions) * 100
```

5 - A confusion matrix is calculated using confusion_matrix from scikit-learn. The confusion matrix shows the counts of true positives, true negatives, false positives, and false negatives.

For each model, a confusion matrix is plotted as a heatmap using sns.heatmap from the Seaborn library. The plot includes annotations, axis labels, and a title with the model name and accuracy.

```
cm = confusion_matrix(y_test, predictions)

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False, ax=axes[i])

axes[i].set_xlabel("Predicted")

axes[i].set_ylabel("Actual")

axes[i].set_title(f"Confusion Matrix - {name} Model\nAccuracy: {accuracy:.2f}%")

axes[i].set_xticks([0.5, 1.5])

axes[i].set_yticks([0.5, 1.5])

axes[i].set_xticklabels(["Class 0", "Class 1"])

axes[i].set_yticklabels(["Class 0", "Class 1"])
```

6. Deep Learning: Neural Network



```python
import tensorflow as tf
from tensorflow import keras

def train_neural_network(X_num, X_cat, y, num_features, X_pca, num_epochs=20, batch_size=32, validation_split=0.2):
    # Determine the number of features based on your dataset
    num_features = X_pca.shape[1]

    # Define the architecture of the neural network with additional hidden layers
    model = keras.Sequential([
        keras.layers.Input(shape=(num_features,)),  # Input layer
        keras.layers.Dense(128, activation='relu'),  # Hidden layer with ReLU activation
        keras.layers.Dense(64, activation='relu'),   # Additional hidden layer with ReLU activation
        keras.layers.Dense(32, activation='relu'),   # Additional hidden layer with ReLU activation
        keras.layers.Dense(16, activation='relu'),   # Additional hidden layer with ReLU activation
        keras.layers.Dense(1, activation='sigmoid')  # Output layer with sigmoid activation for binary classification
    ])

    # Compile the model
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=1)

    # Train the model
    model.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, validation_split=validation_split)

    # Evaluate the model on the test data
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print(f'Test accuracy: {test_acc * 100:.2f}%')

    return model, test_acc

num_features = ['Tenure', 'MonthlyCharges', 'TotalCharges', 'MonthlyToTotalChargesRatio', 'TotalPayment', 'NEW_AVG_Charges',

y = df['Churn']
X_cat = df.drop(columns='Churn')
X_cat = df.drop(columns=num_features)
X_num = df[num_features]

scaler = MinMaxScaler()
X_num_scaled = scaler.fit_transform(X_num)
X_num_scaled = pd.DataFrame(X_num_scaled, columns=num_features)

X = pd.concat([X_cat, X_num_scaled], axis=1)

pca = PCA(n_components = 5)
X_pca = pca.fit_transform(X)

model_nn, test_accuracy = train_neural_network(X_num, X_cat, y, num_features, X_pca)
test_accuracy
```

Fig 6.1: Neural Network code

This code defines and trains a neural network using TensorFlow and Keras for binary classification. It also evaluates the model's accuracy on a test dataset.

1 - Import TensorFlow and Keras libraries

```
import tensorflow as tf

from tensorflow import keras
```

2 - This function is responsible for training a neural network with the specified parameters.

Arguments:

- X_num: Numerical features.
- X_cat: Categorical features (not used in this code).
- y: Target variable (labels).
- num_features: Number of features in the dataset.
- X_pca: Data after PCA transformation.
- num_epochs: Number of training epochs (default is 20).
- batch_size: Batch size for training (default is 32).
- validation_split: Fraction of training data to use for validation (default is 20%).

```
def train_neural_network(X_num, X_cat, y, num_features, X_pca,
num_epochs=20, batch_size=32, validation_split=0.2):
```

3 - Determine the number of features based on the shape of X_pca.

```
num_features = X_pca.shape[1]
```

4 - This code defines a neural network with an input layer (Input) and several hidden layers using ReLU activation functions. The final output layer uses a sigmoid activation function for binary classification.

```
model = keras.Sequential([

     keras.layers.Input(shape=(num_features,)),  # Input layer

     keras.layers.Dense(128, activation='relu'),  # Hidden layer
with ReLU activation

     keras.layers.Dense(64, activation='relu'),   # Additional
hidden layer with ReLU  activation

     keras.layers.Dense(32, activation='relu'),   # Additional
hidden layer with ReLU activation

     keras.layers.Dense(16, activation='relu'),   # Additional
hidden layer with ReLU activation

     keras.layers.Dense(1, activation='sigmoid')  # Output layer
with sigmoid activation for binary classification

    ])
```

5 - The model is compiled with the Adam optimizer, binary cross-entropy loss (common for binary classification), and accuracy as the evaluation metric.

```
model.compile(optimizer='adam',

     loss='binary_crossentropy',

     metrics=['accuracy'])
```

6 - The data is split into training and testing sets using train_test_split from scikit-learn. The testing set is 20% of the data.

The model is trained on the training data for the specified number of epochs, with the specified batch size and validation split.

```
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, 
test_size=0.2, random_state=1)

model.fit(X_train, y_train, epochs=num_epochs, 
batch_size=batch_size, validation_split=validation_split)
```

7 - Evaluate the model on the test data and print the test accuracy.

```
test_loss, test_acc = model.evaluate(X_test, y_test)

print(f'Test accuracy: {test_acc * 100:.2f}%')
```

8 - Call the train_neural_network function to train the neural network and obtain the test accuracy

```
model_nn, test_accuracy = train_neural_network(X_num, X_cat, y, 
num_features, X_pca)
```

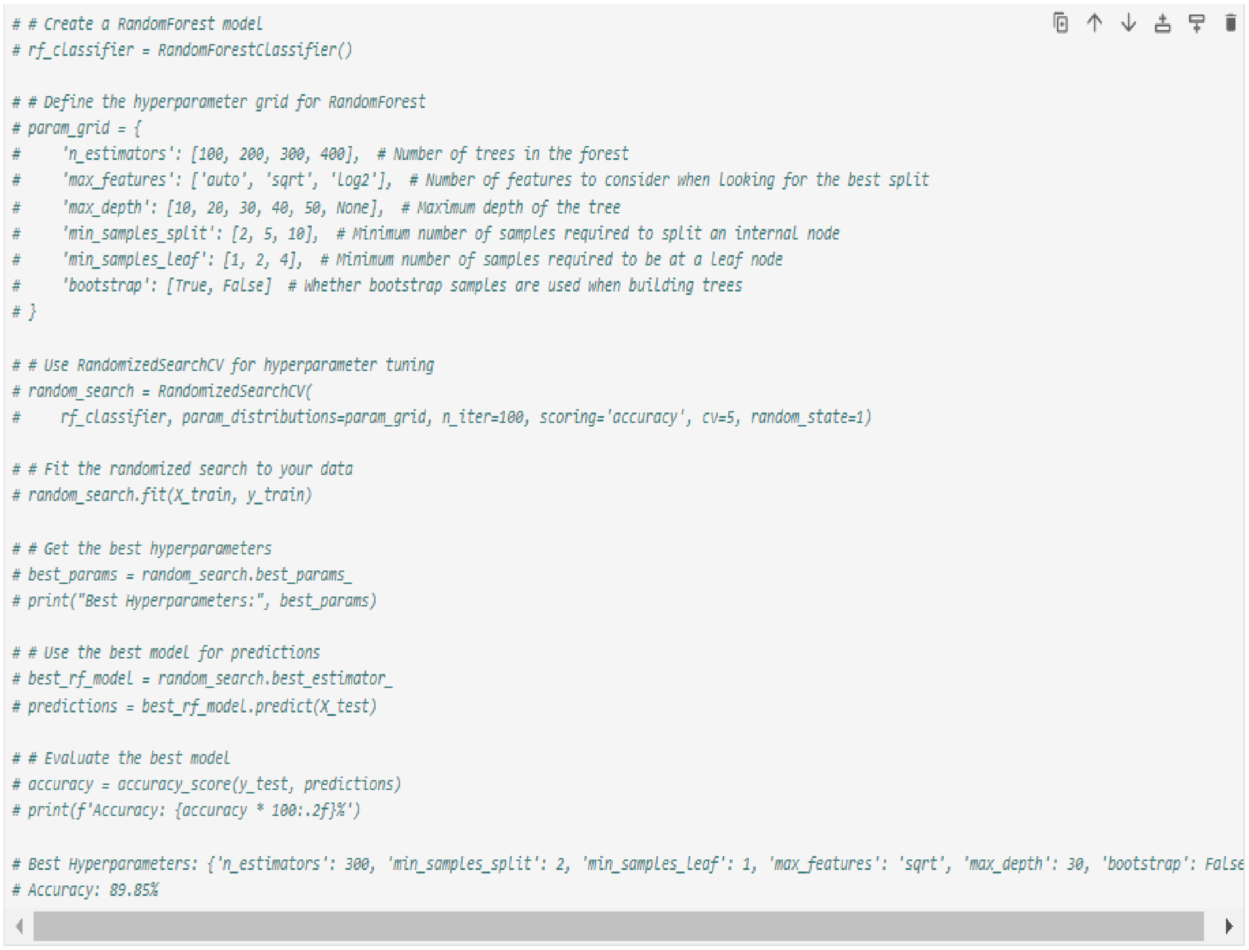
7. Random Forest Classifier Model Evaluation and Confusion Matrix

7.1. Define the best hyperparameters for RandomForestClassifier:

This code performs hyperparameter tuning for a Random Forest classifier using the RandomizedSearchCV approach to find the best combination of hyperparameters for your machine learning model.

This code is commented in the original file because it takes too long to implement but we extracted the best combination of hyperparameters for the machine learning model and used them in our code.

.



```
# # Create a RandomForest model
# rf_classifier = RandomForestClassifier()

# # Define the hyperparameter grid for RandomForest
# param_grid = {
#     'n_estimators': [100, 200, 300, 400],  # Number of trees in the forest
#     'max_features': ['auto', 'sqrt', 'log2'],  # Number of features to consider when looking for the best split
#     'max_depth': [10, 20, 30, 40, 50, None],  # Maximum depth of the tree
#     'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split an internal node
#     'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required to be at a leaf node
#     'bootstrap': [True, False]  # Whether bootstrap samples are used when building trees
# }

# # Use RandomizedSearchCV for hyperparameter tuning
# random_search = RandomizedSearchCV(
#     rf_classifier, param_distributions=param_grid, n_iter=100, scoring='accuracy', cv=5, random_state=1)

# # Fit the randomized search to your data
# random_search.fit(X_train, y_train)

# # Get the best hyperparameters
# best_params = random_search.best_params_
# print("Best Hyperparameters:", best_params)

# # Use the best model for predictions
# best_rf_model = random_search.best_estimator_
# predictions = best_rf_model.predict(X_test)

# # Evaluate the best model
# accuracy = accuracy_score(y_test, predictions)
# print(f'Accuracy: {accuracy * 100:.2f}%')

# Best Hyperparameters: {'n_estimators': 300, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': 30, 'bootstrap': False
# Accuracy: 89.85%
```

Please note that this part is commented because it takes too long to implement

Fig 7.1: RandomizedSearchCV for hyperparameter tuning code

1 - This initializes a Random Forest classifier with default hyperparameters.

```
rf_classifier = RandomForestClassifier()
```

2 - param_grid defines a grid of hyperparameters to search through during hyperparameter tuning. It specifies various options for the number of trees in the forest (n_estimators), the number of features to consider for the best split (max_features), the maximum depth of trees (max_depth), and other hyperparameters.

```
param_grid = {
    'n_estimators': [100, 200, 300, 400],  # Number of trees in the forest
    'max_features': ['auto', 'sqrt', 'log2'],  # Number of features to consider when looking for the best split
    'max_depth': [10, 20, 30, 40, 50, None],  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10],  # Minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required to be at a leaf node
    'bootstrap': [True, False]  # Whether bootstrap samples are used when building trees
}
```

3 - RandomizedSearchCV is a method from scikit-learn that performs a randomized search over specified hyperparameter values. It uses cross-validation (cv=5) to evaluate different combinations of hyperparameters and selects the best set based on the accuracy metric (scoring='accuracy').

n_iter controls the number of random combinations to try from the param_grid.

```
random_search = RandomizedSearchCV(
    rf_classifier, param_distributions=param_grid, n_iter=100, scoring='accuracy', cv=5, random_state=1)
```

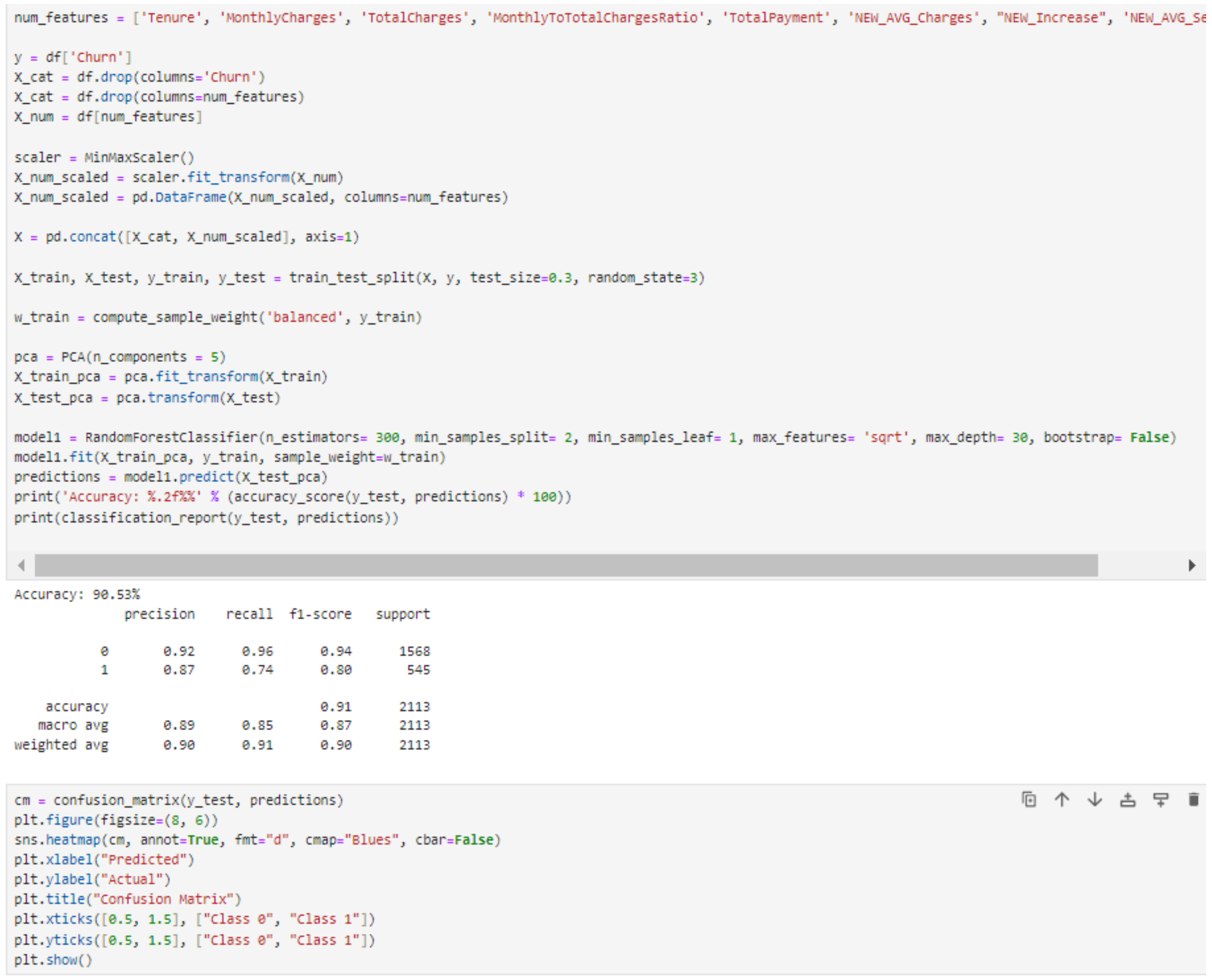
4 - This step fits the randomized search on the training data, allowing it to explore various hyperparameter combinations.

```
random_search.fit(X_train, y_train)
```

5 - After the search is complete, the best hyperparameters are printed.

```
best_params = random_search.best_params_

print("Best Hyperparameters:", best_params)
```

a. Evaluating RandomForestClassifier model:



```
num_features = ['Tenure', 'MonthlyCharges', 'TotalCharges', 'MonthlyToTotalChargesRatio', 'TotalPayment', 'NEW_AVG_Charges', "NEW_Increase", 'NEW_AVG_Se

y = df['Churn']
X_cat = df.drop(columns='Churn')
X_cat = df.drop(columns=num_features)
X_num = df[num_features]

scaler = MinMaxScaler()
X_num_scaled = scaler.fit_transform(X_num)
X_num_scaled = pd.DataFrame(X_num_scaled, columns=num_features)

X = pd.concat([X_cat, X_num_scaled], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=3)

w_train = compute_sample_weight('balanced', y_train)

pca = PCA(n_components = 5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

model1 = RandomForestClassifier(n_estimators= 300, min_samples_split= 2, min_samples_leaf= 1, max_features= 'sqrt', max_depth= 30, bootstrap= False)
model1.fit(X_train_pca, y_train, sample_weight=w_train)
predictions = model1.predict(X_test_pca)
print('Accuracy: %.2f%%' % (accuracy_score(y_test, predictions) * 100))
print(classification_report(y_test, predictions))
```

```
Accuracy: 90.53%
              precision    recall  f1-score   support

           0       0.92      0.96      0.94      1568
           1       0.87      0.74      0.80       545

    accuracy                           0.91      2113
   macro avg       0.89      0.85      0.87      2113
weighted avg       0.90      0.91      0.90      2113
```

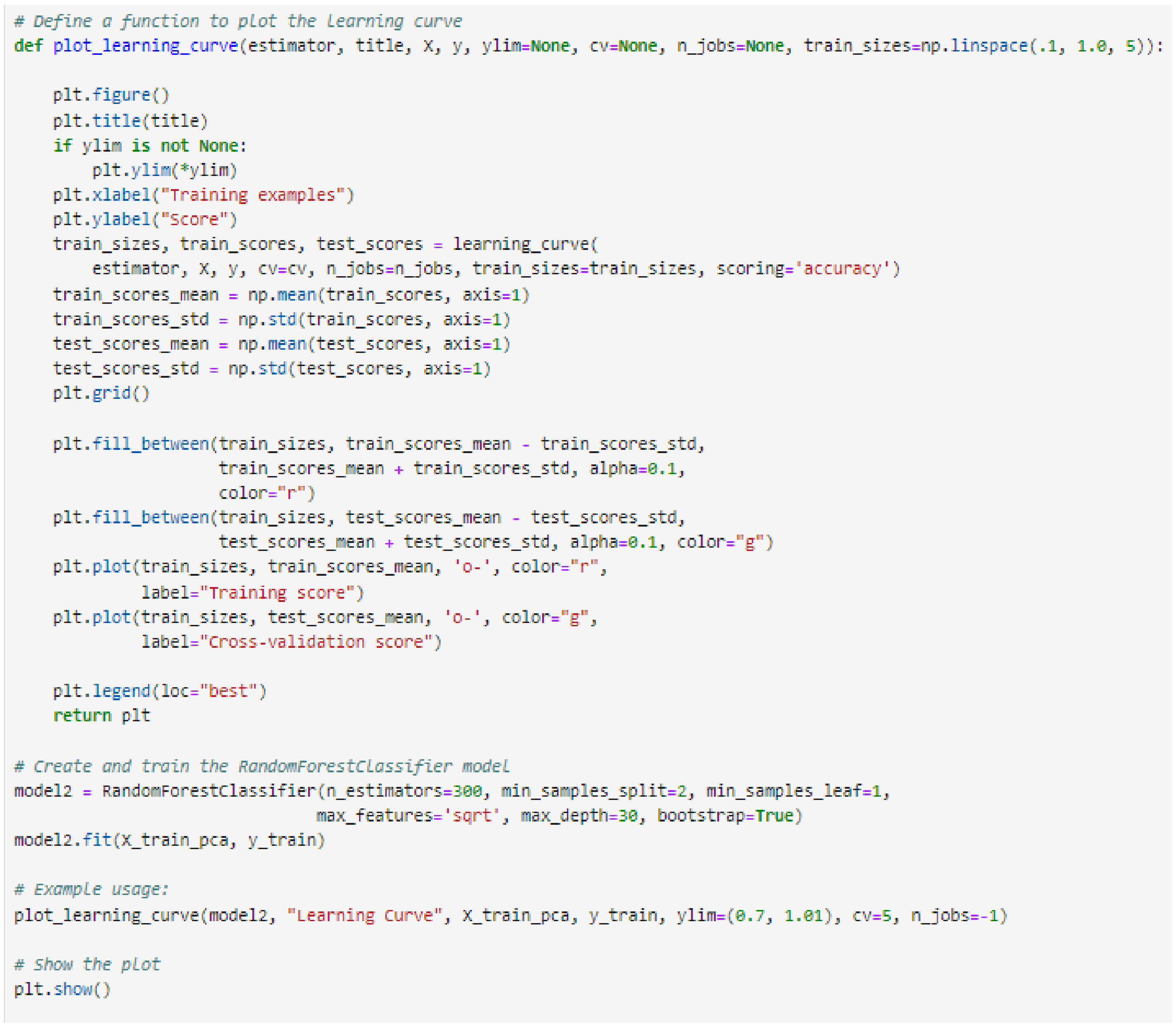
```
cm = confusion_matrix(y_test, predictions)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.xticks([0.5, 1.5], ["Class 0", "Class 1"])
plt.yticks([0.5, 1.5], ["Class 0", "Class 1"])
plt.show()
```

Fig 7.2: RandomForestClassifier model code

In summary, this code preprocesses data, applies PCA for dimensionality reduction, trains a Random Forest classifier with class weighting, evaluates the classifier's performance, and visualizes the confusion matrix to assess the model's performance in a binary classification task.

8. Learning curve

This code defines a function plot_learning_curve and demonstrates how to use it to plot the learning curve for a machine learning model.



```python
# Define a function to plot the learning curve
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None, n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):

    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes, scoring='accuracy')
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt

# Create and train the RandomForestClassifier model
model2 = RandomForestClassifier(n_estimators=300, min_samples_split=2, min_samples_leaf=1,
                               max_features='sqrt', max_depth=30, bootstrap=True)
model2.fit(X_train_pca, y_train)

# Example usage:
plot_learning_curve(model2, "Learning Curve", X_train_pca, y_train, ylim=(0.7, 1.01), cv=5, n_jobs=-1)

# Show the plot
plt.show()
```

Fig 8.1: Learning curve code

1 - Importing learning_curve module from scikit_learn library.

```python
from sklearn.model_selection import learning_curve
```

2 - This function is defined to plot the learning curve of a machine learning model.

Arguments:

- estimator: The machine learning model to be evaluated.
- title: The title for the learning curve plot.
- X: The feature matrix.
- y: The target variable.
- ylim: Optional parameter to set the y-axis limits for the plot.
- cv: Cross-validation splitting strategy (e.g., cv=5 for 5-fold cross-validation).
- n_jobs: Number of CPU cores to use for parallelization (-1 uses all available cores).
- train_sizes: An array of training set sizes used to generate the learning curve.

```
def plot_learning_curve(estimator, title, X, y, ylim=None,
cv=None, n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
```

3 - The code initializes a new Matplotlib figure and sets the title, y-axis label, and (optionally) the y-axis limits.

```
plt.figure()

plt.title(title)

if ylim is not None:

    plt.ylim(*ylim)

plt.xlabel("Training examples")

plt.ylabel("Score")
```

4 - train_sizes: The sizes of the training sets used for training the model.

train_scores: Training scores (accuracy) for each training set size.

test_scores: Cross-validation scores (accuracy) for each training set size.

```
train_sizes, train_scores, test_scores = learning_curve(

     estimator, X, y, cv=cv, n_jobs=n_jobs,

     train_sizes=train_sizes, scoring='accuracy')
```

5 - The mean and standard deviation of training and test scores are calculated along the rows (axis=1) to obtain average scores and score variances.

```
train_scores_mean = np.mean(train_scores, axis=1)

train_scores_std = np.std(train_scores, axis=1)

test_scores_mean = np.mean(test_scores, axis=1)

test_scores_std = np.std(test_scores, axis=1)
```

6 - The learning curve is plotted with shaded regions representing one standard deviation above and below the mean scores. This shading indicates the variability of the scores.

The shaded regions are filled with colors, and the curves for training and cross-validation scores are plotted using circles connected by lines.

```
plt.grid()

plt.fill_between(train_sizes, train_scores_mean -
train_scores_std,

                 train_scores_mean + train_scores_std,
alpha=0.1,

                 color="r")

plt.fill_between(train_sizes, test_scores_mean -
test_scores_std,

                 test_scores_mean + test_scores_std, alpha=0.1,
color="g")

plt.plot(train_sizes, train_scores_mean, 'o-', color="r",

         label="Training score")

plt.plot(train_sizes, test_scores_mean, 'o-', color="g",

         label="Cross-validation score")
```

7 - A Random Forest classifier (model2) is created with specified hyperparameters, and it is trained on the training data (X_train_pca and y_train).

```
model2 = RandomForestClassifier(n_estimators=300,
      min_samples_split=2, min_samples_leaf=1,
      max_features='sqrt', max_depth=30, bootstrap=True)
model2.fit(X_train_pca, y_train)
```

8 - The plot_learning_curve function is called to create a learning curve for the model2 classifier using the training data.

```
plot_learning_curve(model2, "Learning Curve", X_train_pca,
y_train, ylim=(0.7, 1.01), cv=5, n_jobs=-1)
```

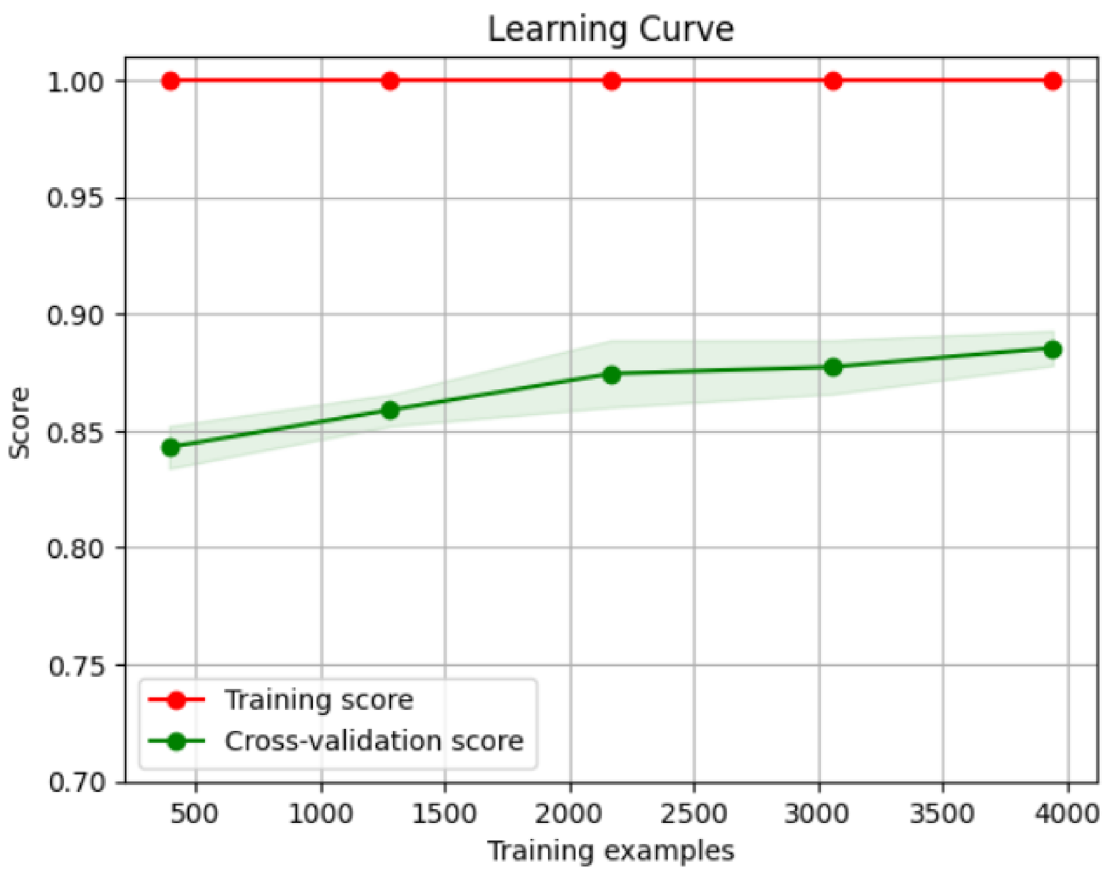
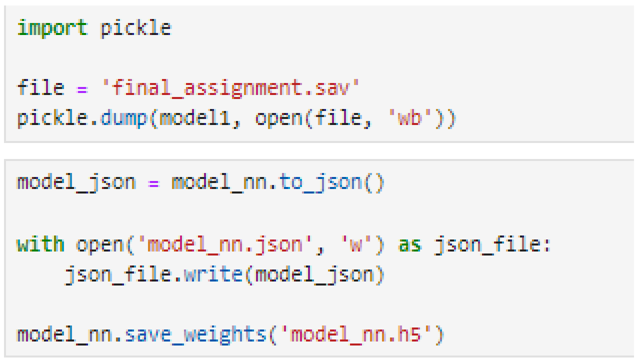


Fig 8.2: Learning curve plot

9. Saving the models

Saving the RandomForestClassifier model using pickle library and saving Neural Network model as a json file with its weights.



```
import pickle

file = 'final_assignment.sav'
pickle.dump(model1, open(file, 'wb'))
```

```
model_json = model_nn.to_json()

with open('model_nn.json', 'w') as json_file:
    json_file.write(model_json)

model_nn.save_weights('model_nn.h5')
```

Fig 9.1: Saving the model plot

10. GUI Application And Predictions

This code is for creating a graphical user interface (GUI) application using the Tkinter library in Python. The purpose of this application is to predict customer churn based on user input for various features related to telecommunications services.

1 - Import necessary libraries for creating the GUI and processing user input.

```
import tkinter as tk
from tkinter import ttk
import pandas as pd
import joblib
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.decomposition import PCA
from tensorflow.keras.models import model_from_json
```

2 - Create the main application window using Tkinter.

```
window = tk.Tk()

window.geometry("480x700")

window.title("Churn Prediction App")
```

3 - Create a vertical scrollbar and a canvas to hold the elements of the GUI. The scrollbar is used to scroll through the contents of the canvas.

```
scrollbar = tk.Scrollbar(window)

scrollbar.pack(side="right", fill="y")


canvas = tk.Canvas(window, yscrollcommand=scrollbar.set)

canvas.pack(fill="both", expand=True)
```

.4 - Configure the scrollbar to work with the canvas.

```
scrollbar.config(command=canvas.yview)
```

5 - Create a frame to hold the GUI elements. The frame will be placed inside the canvas.

```
frame = tk.Frame(canvas)

canvas.create_window((0, 0), window=frame, anchor="nw")
```

6 - Create labels and input fields (both text and dropdown) for various features. The feature_labels list contains the names of features to be displayed.

```
feature_labels = [

    "Gender", "SeniorCitizen", "Partner", "Dependents",
"Tenure", "PhoneService", "MultipleLines",

    "InternetService", "OnlineSecurity", "OnlineBackup",
"DeviceProtection", "TechSupport", "StreamingTV",

    "StreamingMovies", "Contract", "PaperlessBilling",
"PaymentMethod", "MonthlyCharges", "TotalCharges"
]


entry_fields = {}      # Dictionary to hold Entry widgets for
numeric features

dropdown_fields = {}   # Dictionary to hold Combobox widgets for
categorical features

userSelection = [""] * len(feature_labels)  # Initialize
userSelection with empty strings


# Create labels and input fields for each feature

for idx, feature_label in enumerate(feature_labels):

    # Create labels

    tk.Label(frame, text=feature_label + ":").grid(row=3 + idx,
column=0, pady=(5, 5))


    # Create input fields for numerical features

    if feature_label in ["MonthlyCharges", "TotalCharges",
"Tenure"]:

        entry_fields[feature_label] = tk.Entry(frame,
font=("Arial", 12))

        entry_fields[feature_label].grid(row=3 + idx, column=1,
pady=(5, 5))

        entry_fields[feature_label].bind("<FocusOut>", lambda
event, label=feature_label: update_user_selection_entry(event,
label))

        numerical_fields.append(entry_fields[feature_label])


    # Create a dropdown list for features with predefined
options

    elif feature_label in feature_options:

        feature_values = feature_options[feature_label]

        feature_var = tk.StringVar()
        feature_combobox = ttk.Combobox(frame,
textvariable=feature_var, values=feature_values)

        feature_var.trace_add("write", lambda *args,
var=feature_var, label=feature_label: update_user_selection(var,
label))

        feature_combobox.grid(row=3 + idx, column=1,
columnspan=3, pady=(5, 5))

        dropdown_fields[feature_label] = feature_combobox
```

7 - Define two functions (update_user_selection and update_user_selection_entry) to update the userSelection list when the user selects options from dropdowns or enters numeric values in the entry fields.

```
def update_user_selection(var, label):

    selected_value = var.get()

    userSelection[feature_labels.index(label)] = selected_value


def update_user_selection_entry(event, label):

    entry_value = event.widget.get()

    if entry_value.strip():

        userSelection[feature_labels.index(label)] =
float(entry_value)

    else:

        userSelection[feature_labels.index(label)] = ""
```

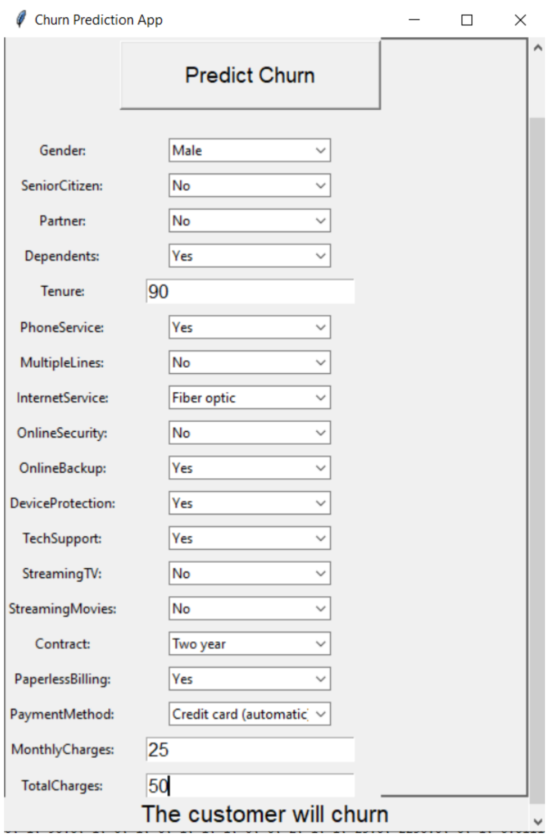
8 - Create a button called "Predict Churn" that, when clicked, invokes the predict_and_display_result function.

```
predict_button = tk.Button(frame, text='Predict Churn',
command=predict_and_display_result, height=2, width=20,
font=("Arial", 14))

predict_button.grid(row=2, column=1, pady=(10, 20))
```

9 - The predict_and_display_result Function:

- This function is called when the "Predict Churn" button is clicked. It performs the following steps:
- Transform user input, including label encoding for categorical features and feature engineering.
- Load a trained neural network model from JSON and weights.
- Use the model to make predictions and display the result in the GUI.

```
def on_frame_configure(event):

    canvas.configure(scrollregion=canvas.bbox("all"))


frame.bind("<Configure>", on_frame_configure)


transformed_values = []  # Initialize an empty list to store transformed values

# Start the Tkinter main loop

window.mainloop()
```



Gender:	Male
SeniorCitizen:	No
Partner:	No
Dependents:	Yes
Tenure:	90
PhoneService:	Yes
MultipleLines:	No
InternetService:	Fiber optic
OnlineSecurity:	No
OnlineBackup:	Yes
DeviceProtection:	Yes
TechSupport:	Yes
StreamingTV:	No
StreamingMovies:	No
Contract:	Two year
PaperlessBilling:	Yes
PaymentMethod:	Credit card (automatic
MonthlyCharges:	25
TotalCharges:	50

The customer will churn

Fig 10.1: Churn prediction app

11. Summary of Churn Prediction GUI Project:

Overview:

The project features a GUI interface to predict customer churn in a telecom service. Users input customer data, and the system predicts churn based on a pre-trained neural network model.

Components:

User Interface: A user-friendly GUI for inputting data.

Feature Inputs: Users provide telecom service and customer details.

Data Processing: Input data undergoes encoding and feature engineering.

Neural Network Model: A pre-trained model makes churn predictions.

Prediction: Users receive churn predictions based on their input.

Data Saving: User data and predictions are saved to CSV files.

Visualization: The GUI displays input fields and prediction results.

Enhancements:

Incorporate advanced ML models.

Improve user interface.

Add more features for insights.

The Churn Prediction GUI aids customer retention efforts by predicting churn based on user-provided data.