Deep Learning Multi-Class Classification Using Keras

This presentation explores the process of building and evaluating a multi-class classification model using Keras, a popular deep learning library. We will compare the performance of a best practices model with a model designed using Keras Tuner.

Krishna Khandelwal

1. **Importing Key Libraries**: You imported essential libraries for data manipulation (`pandas`, `numpy`), machine learning (`scikit-learn` for model selection, preprocessing, etc.), and deep learning (`tensorflow` and `keras` for neural network modeling).

2. **Data Preprocessing**: You brought in tools like `train_test_split` (to split data into training and test sets), `StandardScaler` (for feature scaling), and `SimpleImputer` (for handling missing values).

3. **Visualization**: You installed and imported `missingno` and `plotly.express` to visualize missing data and generate plots. You also used `matplotlib` for additional plotting.

4. **Modeling and Tuning**: You imported Keras libraries for building and visualizing models (`plot_model`), and tools like `keras_tuner` for hyperparameter tuning, allowing you to optimize the model with techniques such as `RandomSearch` and `BayesianOptimization`.

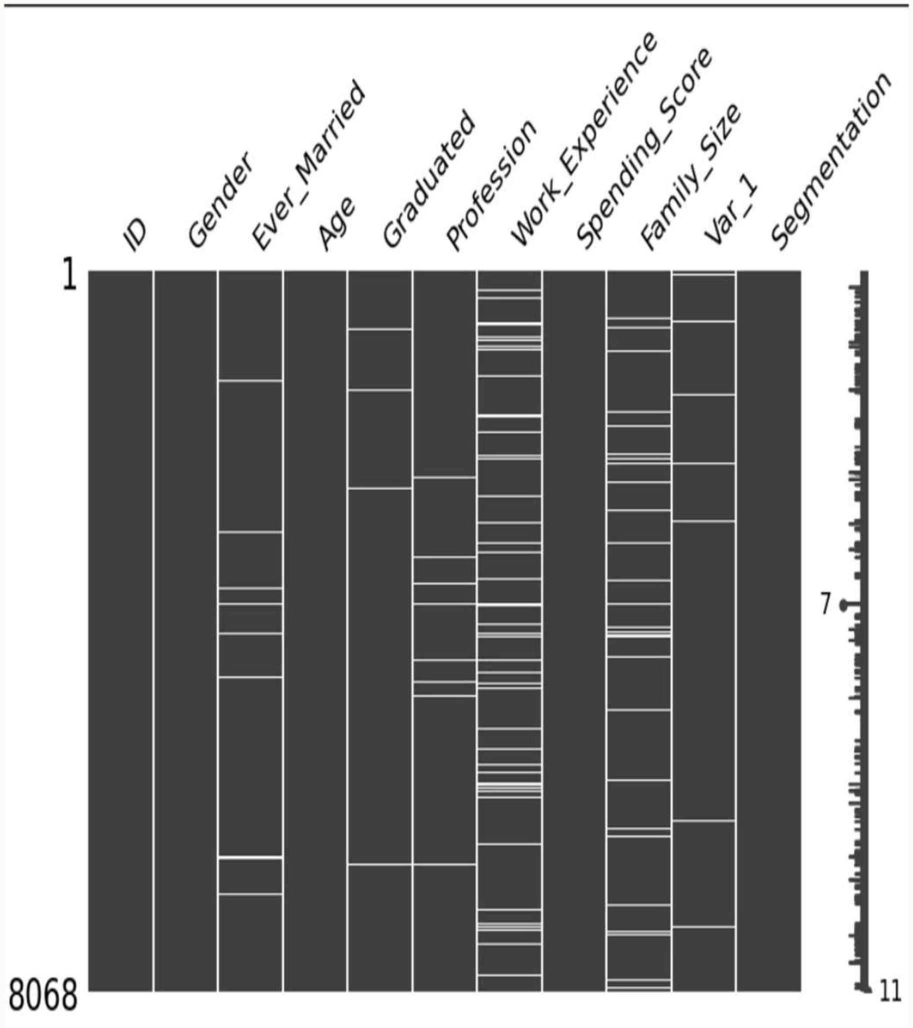
5. **Performance Evaluation**: You included metrics such as `classification_report` and `confusion_matrix` from `sklearn` to evaluate model accuracy and classification performance.

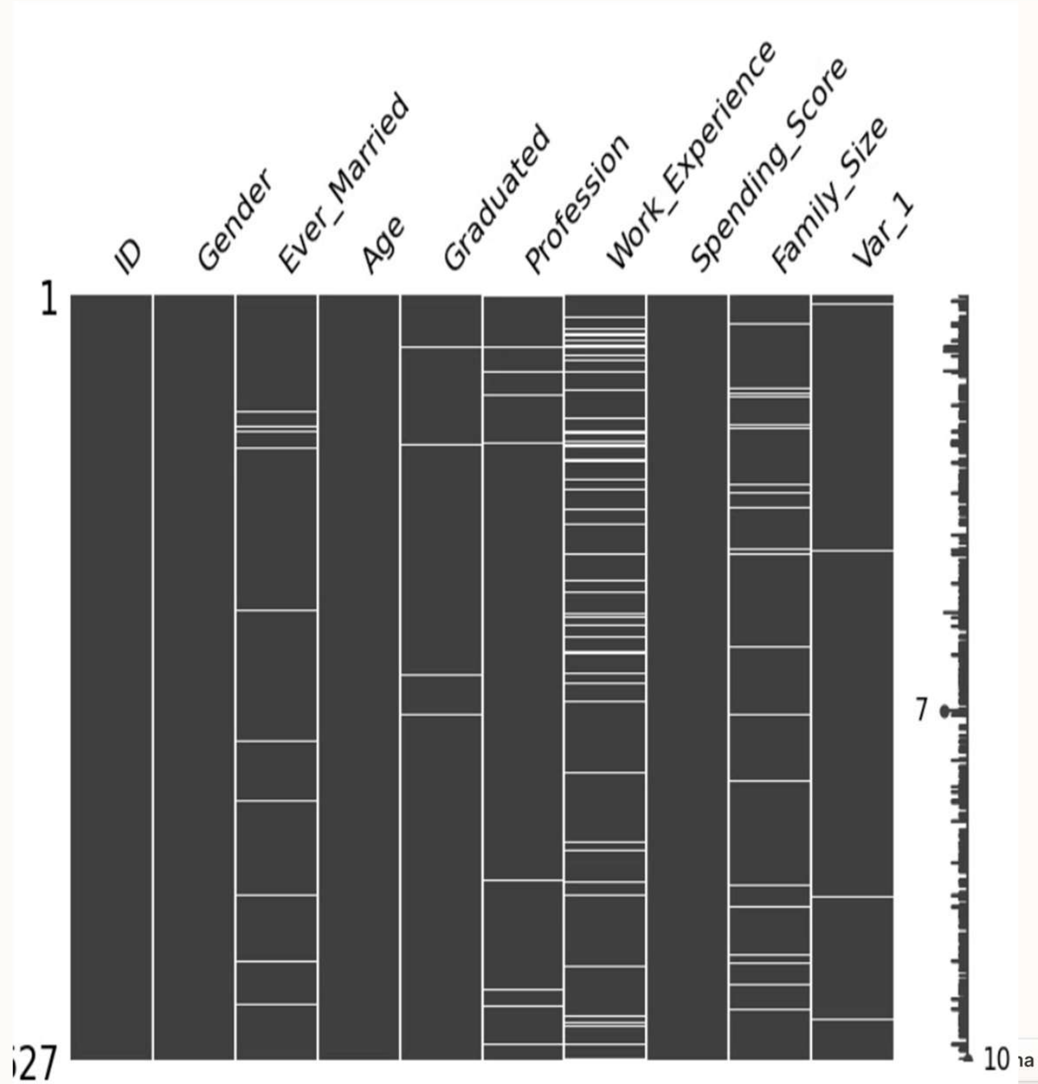
```python
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.utils import to_categorical
import numpy as np
!pip install missingno
!pip install plotly.express
import missingno as msno
import matplotlib.pyplot as plt
import plotly.express as px
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from keras.utils import plot_model
from IPython.display import Image
from sklearn.metrics import classification_report, accuracy_score

from keras_tuner.engine.hyperparameters import HyperParameters
from keras_tuner import RandomSearch, Hyperband, BayesianOptimization
from tensorflow.keras.utils import plot_model
from tensorflow.keras.callbacks import LearningRateScheduler, TensorBoard
import keras_tuner as kt
from sklearn.metrics import classification_report, confusion_matrix

```

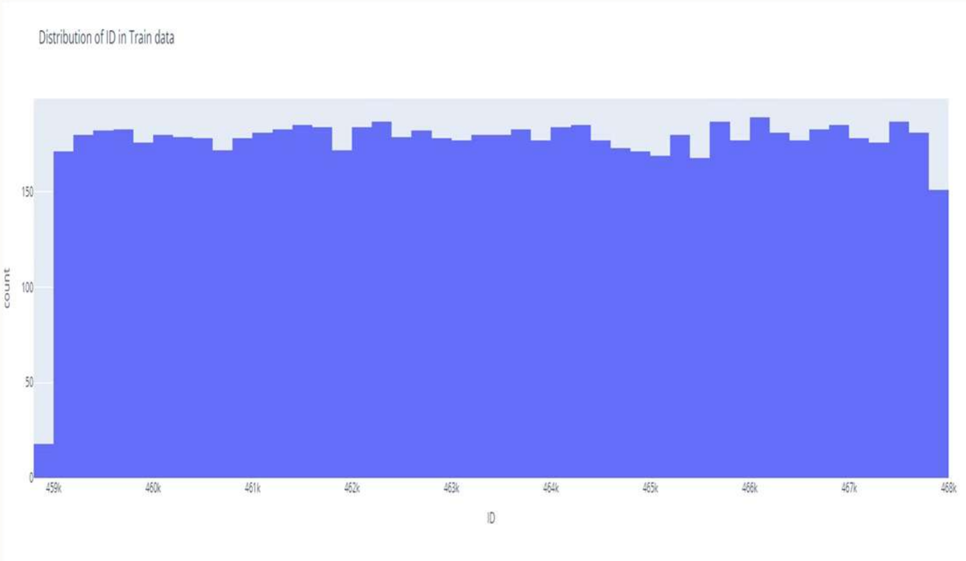
```
[132]
# Define the paths for train and test datasets in your Google Drive
train_path = '/content/drive/MyDrive/Colab Notebooks/train.csv'
test_path = '/content/drive/MyDrive/Colab Notebooks/test.csv'

# Load the datasets
import pandas as pd
train_df = pd.read_csv(train_path)
test_df = pd.read_csv(test_path)

[133]
# Check for missing values in train and test data
print(train_df.isnull().sum())

ID                   0
Gender               0
Ever_Married       140
Age                  0
Graduated           78
Profession         124
Work_Experience    829
Spending_Score       0
Family_Size        335
Var_1               76
Segmentation         0
dtype: int64

[134]
print(test_df.isnull().sum())

ID                   0
Gender               0
Ever_Married        50
Age                  0
Graduated           24
Profession          38
Work_Experience    269
Spending_Score       0
Family_Size        113
Var_1               32
dtype: int64
```

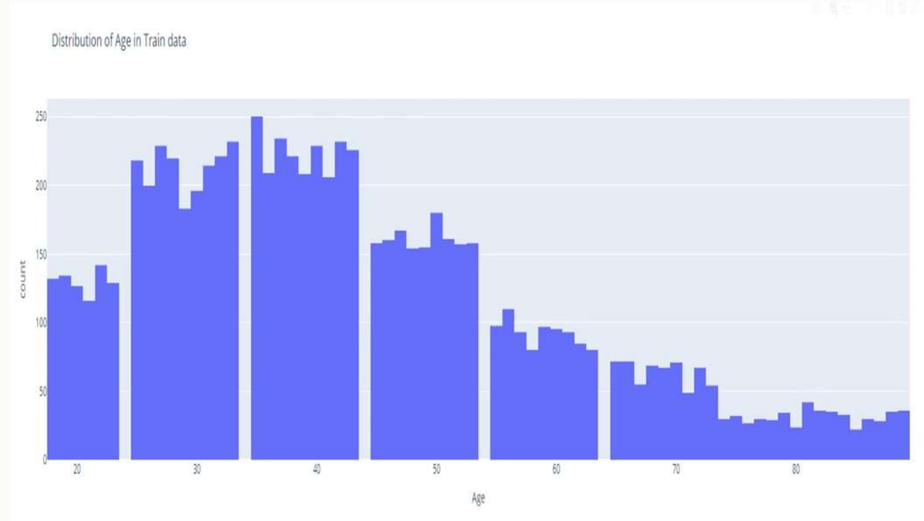
- **Visualizing Missing Values**:Using the `missingno.matrix()` function, you're visualizing missing data in both the training (`train_df`) and test datasets (`test_df`). The `msno.matrix()` function gives a matrix plot showing the presence or absence of data for each column.
- You called `plt.show()` to display the plot for both datasets.
- **Plotting Data Distributions**:
- I looped through all the numerical columns (of types `float64` and `int64`) in both the `train_df` and `test_df` datasets. For each column, you created a histogram using `plotly.express` (`px.histogram`) to visualize the distribution of each feature in both datasets.
- Each histogram is dynamically generated with a title to indicate which column's distribution is being shown, and `fig.show()` renders the plot.

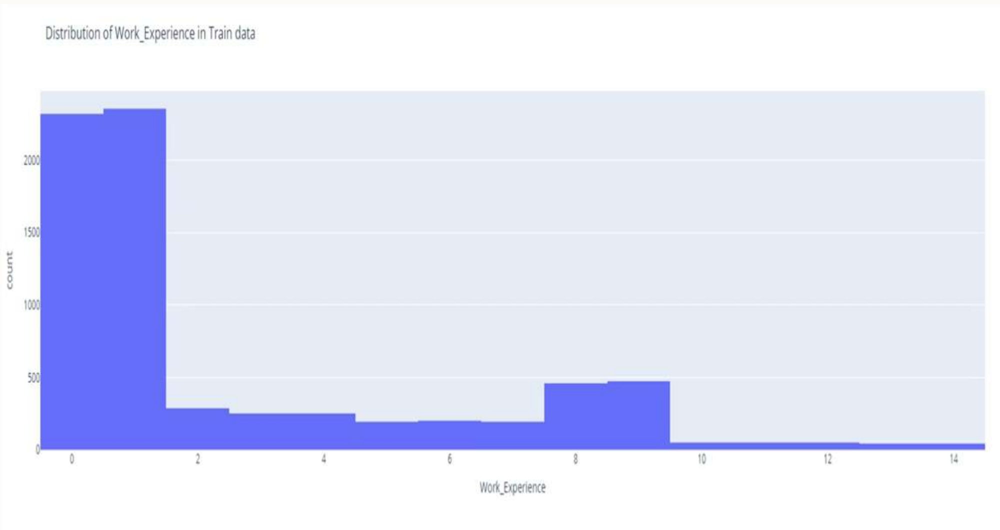
```
[138]
!pip install missingno
!pip install plotly.express
import missingno as msno
import matplotlib.pyplot as plt
import plotly.express as px
# Visualize missing values
msno.matrix(train_df, figsize=(10, 6)) #removed extra indentation
plt.show() #removed extra indentation
msno.matrix(test_df, figsize=(10, 6)) #removed extra indentation
plt.show() #removed extra indentation

# Plot distributions of numerical columns
for column in train_df.select_dtypes(include=['float64', 'int64']).columns:
    fig = px.histogram(train_df, x=column, title=f'Distribution of {column} in Train data')
    fig.show()
for column in test_df.select_dtypes(include=['float64', 'int64']).columns:
    fig = px.histogram(test_df, x=column, title=f'Distribution of {column} in Test data')
    fig.show()
```

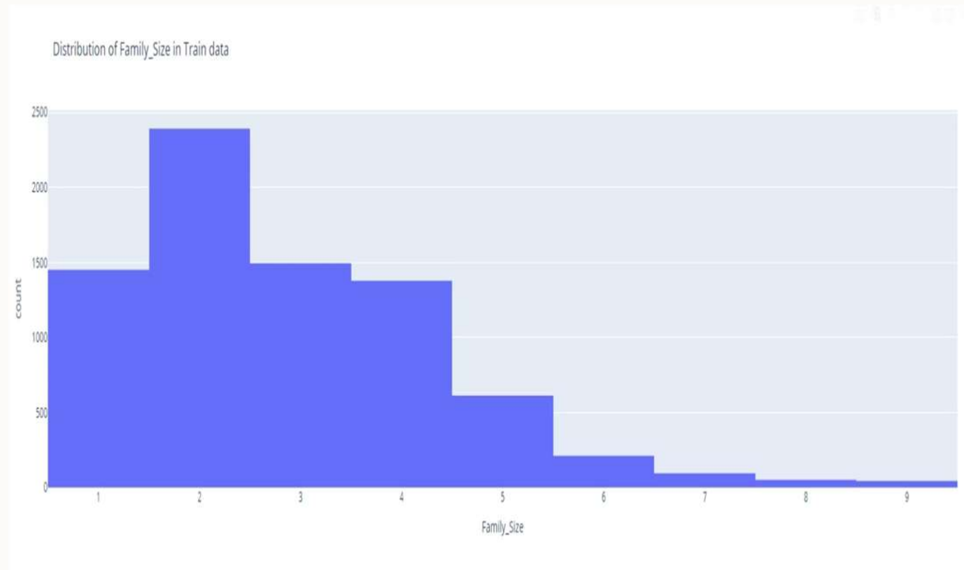

ID
Gender
Ever_Married
Age
Graduated
Profession
Work_Experience
Spending_Score
Family_Size
Var_1
Segmentation
1
8068
7
11


ID
Gender
Ever_Married
Age
Graduated
Profession
Work_Experience
Spending_Score
Family_Size
Var_1
1
7
10


Distribution of ID in Train data
count
ID

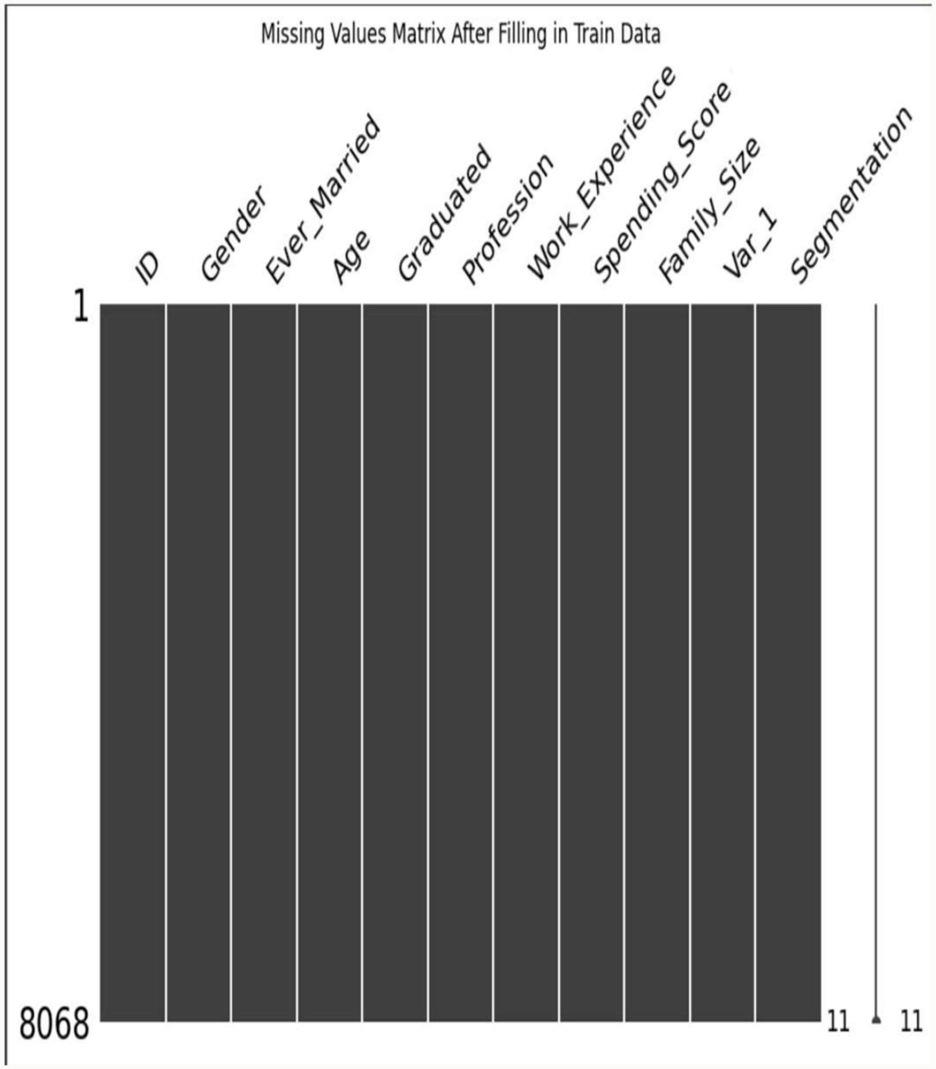

Distribution of Age in Train data
count
Age

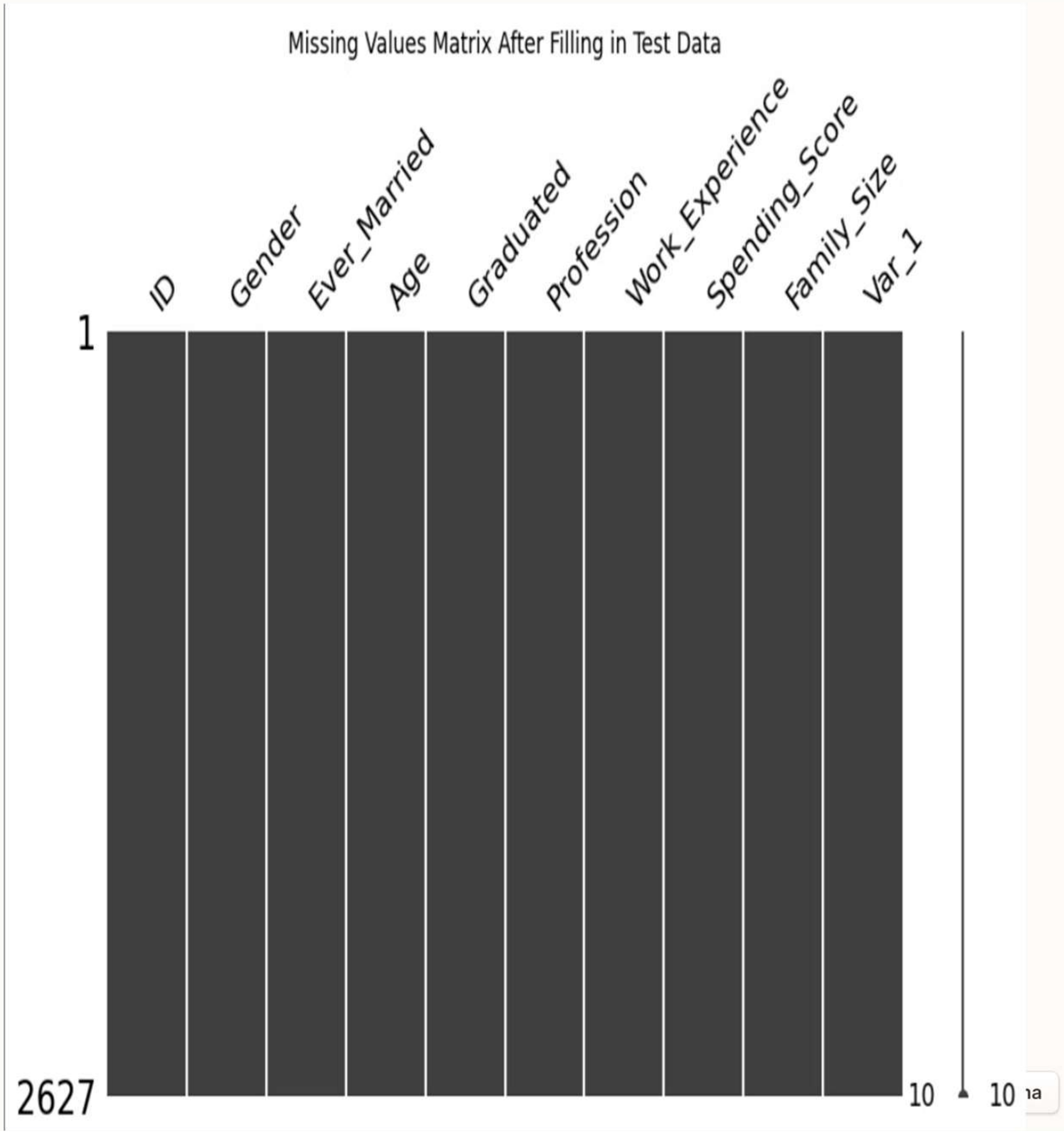

Distribution of Work_Experience in Train data
count
Work_Experience


Distribution of Family_Size in Train data
count
Family_Size

1. **Copied the original datasets** into new variables `train_df_copy` and `test_df_copy` to preserve the originals.
2. **Filled missing values for categorical variables** in both datasets using the **mode** of each column.
3. **Handled missing values for numerical variables** by filling them with the **median** of each respective column.
4. **Visualized missing data post-imputation** using `missingno.matrix()` for both the modified train and test datasets, ensuring missing data was addressed.
5. **Displayed missing values matrix** with a clear title for both datasets after filling, confirming the changes visually.

```python
# Copy the original datasets to new variables
train_df_copy = train_df.copy()
test_df_copy = test_df.copy()

# Handle missing values for categorical variables (mode)
for df in [train_df_copy, test_df_copy]:
    categorical_columns = df.select_dtypes(include=['object']).columns
    for column in categorical_columns:
        df[column].fillna(df[column].mode()[0], inplace=True)

# Handle missing values for numerical variables (median)
for df in [train_df_copy, test_df_copy]:
    numerical_columns = df.select_dtypes(include=['float64', 'int64']).columns
    for column in numerical_columns:
        df[column].fillna(df[column].median(), inplace=True)

# Visualize missing values for both train and test datasets
# Train dataset visualization
msno.matrix(train_df_copy, figsize=(10, 6))
plt.title('Missing Values Matrix After Filling in Train Data')
plt.show()

# Test dataset visualization
msno.matrix(test_df_copy, figsize=(10, 6))
plt.title('Missing Values Matrix After Filling in Test Data')
plt.show()

```


Missing Values Matrix After Filling in Train Data
ID
Gender
Ever_Married
Age
Graduated
Profession
Work_Experience
Spending_Score
Family_Size
Var_1
Segmentation
1
8068
11
11


Missing Values Matrix After Filling in Test Data
ID
Gender
Ever_Married
Age
Graduated
Profession
Work_Experience
Spending_Score
Family_Size
Var_1
1
2627
10
10

```python
categorical_columns = X_train.select_dtypes(include=['object']).columns  # Categorical columns
numerical_columns = X_train.select_dtypes(include=['float64', 'int64']).columns  # Numerical columns

#Create a preprocessor that applies OneHotEncoder to categorical columns and StandardScaler to numerical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='median')),  # Impute missing numerical data with median
            ('scaler', StandardScaler())  # Scale numerical data
        ]), numerical_columns),

        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing categorical data with most frequent
            ('encoder', OneHotEncoder(drop='first'))  # OneHotEncode categorical data
        ]), categorical_columns)
    ])

#Apply the preprocessor to X_train and fit the transformer
X_train_prepared = preprocessor.fit_transform(X_train)

#Apply the same preprocessor to the test data (X_test)
X_test_prepared = preprocessor.transform(test_data)  # Only transform, do not fit on test data

#Check the shapes of the prepared data
print(f'Shape of X_train_prepared: {X_train_prepared.shape}')
print(f'Shape of X_test_prepared: {X_test_prepared.shape}')

```

```
Shape of X_train_prepared: (8068, 23)
Shape of X_test_prepared: (2627, 23)
```

```python
# One-hot encode the target labels
y_train_transformed = pd.get_dummies(y_train).values

# Display the shape of the encoded labels
print("Shape of y_train_transformed:", y_train_transformed.shape)
```

```
Shape of y_train_transformed: (8068, 4)
```

1. **Identified categorical and numerical columns** in `X_train` using `select_dtypes()`, storing them in `categorical_columns` and `numerical_columns` respectively.
2. **Created a preprocessor** using `ColumnTransformer` to handle both categorical and numerical columns:
   - **Numerical columns** are imputed using the median and scaled with `StandardScaler`.
   - **Categorical columns** are imputed using the most frequent value and one-hot encoded using `OneHotEncoder`.
3. **Applied the preprocessor** to `X_train` by fitting and transforming the data and then **transformed** `X_test` using the same preprocessor (without fitting on the test data).
4. **Checked the shape of the transformed data** for both `X_train` and `X_test` to confirm that preprocessing was applied correctly.
5. **One-hot encoded the target labels (**`y_train`**)** using `pd.get_dummies()` and printed the shape of the transformed labels to verify the encoding.

1. **Defined a Sequential neural network model**:
   - **First hidden layer**: A `Dense` layer with 64 units, ReLU activation, and input shape based on the number of features in `X_train_prepared`.
   - **Second hidden layer**: A `Dense` layer with 32 units and ReLU activation.
   - **Output layer**: A `Dense` layer with units equal to the number of classes in `y_train_transformed`, using softmax activation for multiclass classification.
2. **Compiled the model** using the **Adam optimizer** and **categorical crossentropy loss** (appropriate for multiclass classification). The model's performance will be evaluated using **accuracy** as a metric.

Model 1 - Best Practices Model

Build a neural network model with at least two hidden layers using the ReLU activation function.

Use softmax as the activation function for the output layer to handle multi-class classification.

Use categorical cross-entropy as the loss function and Adam optimizer.

[112]
```
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_prepared.shape[1],)),  # First hidden layer
    Dense(32, activation='relu'),  # Second hidden layer
    Dense(y_train_transformed.shape[1], activation='softmax')  # Output layer
])

# Step 4: Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Model Training:

Train the model using the train.csv data (X_train, y_train) for 50 epochs with a batch size of 32.

Use early stopping to prevent overfitting (monitor the validation loss and set a patience of 5 epochs).

Use 20% of the training data as a validation set during training

[113]

```
# Import necessary libraries
from tensorflow.keras.callbacks import EarlyStopping

# Step 1: Define early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

# Step 2: Train the model with early stopping
history = model.fit(
    X_train_prepared,
    y_train_transformed,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)
```

```
# Print the model architecture summary
model.summary()
# Show the model structure
plot_model(model, to_file='model_1_structure.png', show_shapes=True, show_layer_names=True)
from IPython.display import Image
Image(filename='model_1_structure.png')
```

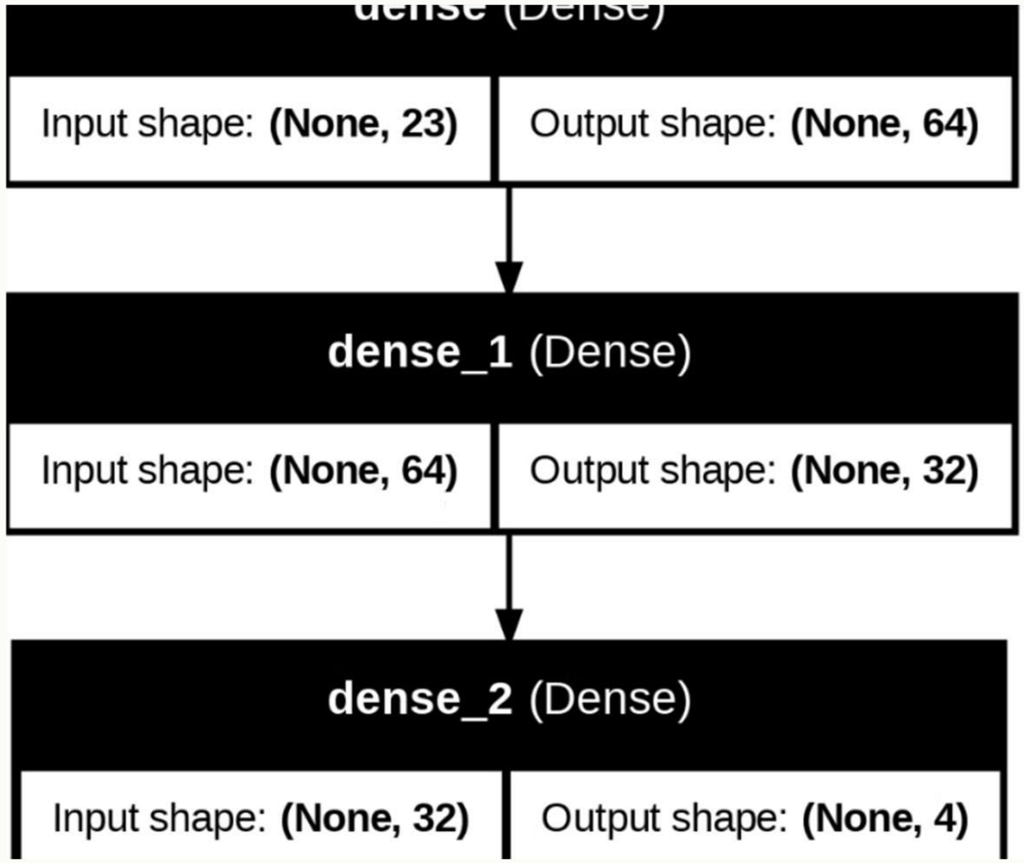
Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	1,536
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 4)	132

Total params: 11,246 (43.93 KB)
Trainable params: 3,748 (14.64 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 7,498 (29.29 KB)

1. **Imported the `EarlyStopping` callback** from `tensorflow.keras.callbacks` to monitor validation loss and stop training if it doesn't improve for 5 epochs (patience of 5), while restoring the best weights.
2. **Defined the early stopping callback** by setting it to monitor the `val_loss` and restoring the best weights when training stops.
3. **Trained the model** on the training data (`X_train_prepared`, `y_train_transformed`) for up to 50 epochs, using a batch size of 32.
    - The training data as a **validation set** (`validation_split=0.2`).
    - **Early stopping** was used to avoid overfitting by stopping the training when validation loss stops improving.
4. **Printed the model summary** to review the architecture and layers of the model, confirming the structure before training.

This approach ensures efficient training while minimizing overfitting.


dense (Dense)
Input shape: (None, 23)
Output shape: (None, 64)
dense_1 (Dense)
Input shape: (None, 64)
Output shape: (None, 32)
dense_2 (Dense)
Input shape: (None, 32)
Output shape: (None, 4)

Model 2 - Hyperparameter Optimization using Keras Tuner:

Model 2 - Hyperparameter Optimization Using Keras Tuner

Implement Keras Tuner:

Use Keras Tuner to optimize the following hyperparameters:

- Number of hidden layers.
- Number of neurons in each layer.
- Learning rate.
- Activation functions.

Set up a search space that explores different combinations of these hyperparameters. Experiment =5

[117]

```
import keras_tuner as kt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

def build_model(hp):
    model_2 = Sequential()

    # Define the input layer based on the shape of X_train
    input_shape = X_train_transformed.shape[1]

    # Tune the number of hidden layers from 1 to 10
    for i in range(hp.Int('num_layers', 1, 10)):

        model_2.add(Dense(units=hp.Int(f'units_{i}', min_value=32, max_value=256, step=32),
                    activation='relu'))

    model_2.add(Dense(4, activation='softmax'))

    learning_rate = hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])

    # Compile the model
    model_2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

    return model_2
```

```
# Initialize the RandomSearch tuner
tuner = kt.RandomSearch(
        build_model,
    objective='val_accuracy',
    max_trials=5,  # Experiment with 5 trials
    executions_per_trial=3,
    directory='keras_tuner_dir',
    project_name='hyperparameter_tuning'
)
```

[119]

```
best_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	1,536
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 4)	132

Total params: 11,246 (43.93 KB)
Trainable params: 3,748 (14.64 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 7,498 (29.29 KB)

1. **Defined a function (`build_model`)** for Keras Tuner to explore the search space:
    - You defined a **sequential model**.
    - Tuned the **number of hidden layers** (from 1 to 10).
    - Tuned the **number of neurons** in each hidden layer (from 32 to 256 with a step of 32).
    - Added the output layer with softmax activation for multiclass classification.
    - Tuned the **learning rate** with options `[1e-2, 1e-3, 1e-4]`.
2. **Set up the Keras Tuner search**:
    - You used `tuner.search()` to explore different combinations of hyperparameters by training the model on the **training data (`X_train_transformed` and `y_train_encoded`)** for 20 epochs, with early stopping and a 20% validation split.
3. **Retrieved the best hyperparameters**:
    - After the search, you retrieved the **best hyperparameters** found by the tuner using `tuner.get_best_hyperparameters()`.
4. **Built and trained the best model**:
    - You constructed the best model (`best_model`) using the optimal hyperparameters and trained it on the training data for 50 epochs with early stopping, using a batch size of 32 and a 20% validation split.

This approach allows you to automate the process of finding the best combination of hyperparameters and optimizes the model for better performance.

1. **Evaluated the best model**:
    - You evaluated the performance of the **best model** (found using Keras Tuner) on the training data (`X_train_prepared`, `y_train_transformed`) to obtain the **training loss** and **accuracy**.
2. **Printed the training metrics**:
    - The training accuracy and loss were printed with formatting to display up to four decimal places for clarity (`train_accuracy:.4f`, `train_loss:.4f`).
3. **Saved the best model**:
    - You saved the trained **best model** using the `.save()` method, which stores the model in an HDF5 file (`best_model.h5`) for future use, allowing you to reload it without retraining.

This ensures you can evaluate and preserve the model for further predictions or deployment.

```
train_loss, train_accuracy = best_model.evaluate(X_train_prepared, y_train_transformed)
print(f"Best Model Training Accuracy: {train_accuracy:.4f}")
print(f"Best Model Training Loss: {train_loss:.4f}")
# Save the model for future use
best_model.save('best_model.h5')
```

```
253/253 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5741 - loss: 0.9776
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.sav
Best Model Training Accuracy: 0.5636
Best Model Training Loss: 0.9951
```

Model Evaluation and Performance

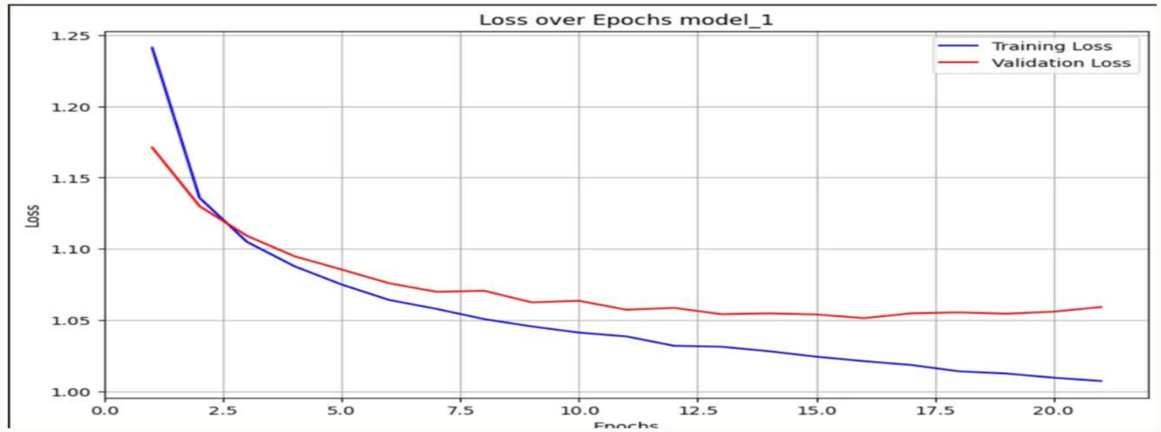
Evaluate both models on the training data using accuracy as the metric.

Generate and print the classification report showing precision, recall, and F1- score for each customer segment.

[121]

```
train_loss, train_accuracy = model.evaluate(X_train_prepared, y_train_encoded, verbose=0)
print(f"Training Accuracy: {train_accuracy:.4f}")

# Step 2: Generate predictions on the training data
y_train_pred_encoded = model.predict(X_train_prepared)

y_train_pred = np.argmax(y_train_pred_encoded, axis=1)

# Step 4: Convert the true one-hot encoded labels to class labels
y_train_true = np.argmax(y_train_encoded, axis=1)

class_labels = label_encoder.classes_  # Assuming label_encoder was used to encode y_train

print("\nClassification Report:")
print(classification_report(y_train_true, y_train_pred, target_names=class_labels))

# Optional: Print accuracy score for verification
print(f"Accuracy Score: {accuracy_score(y_train_true, y_train_pred):.4f}")
```

```
Training Accuracy: 0.5636
253/253 ━━━━━━━━━━━━ 1s 3ms/step

Classification Report:
              precision    recall  f1-score   support

           A       0.49      0.53      0.51      1972
           B       0.47      0.31      0.37      1858
           C       0.57      0.61      0.59      1970
           D       0.66      0.76      0.71      2268

    accuracy                           0.56      8068
   macro avg       0.55      0.55      0.54      8068
weighted avg       0.55      0.56      0.55      8068

Accuracy Score: 0.5636
```



Training vs. Validation Loss Plot

[122]

```
training_loss = history.history['loss']
validation_loss = history.history['val_loss']

# Step 2: Create a range of epochs
epochs_range = range(len(training_loss))

# Step 3: Plot the Training vs. Validation Loss
plt.figure(figsize=(8, 6))
plt.plot(epochs_range, training_loss, label='Training Loss')
plt.plot(epochs_range, validation_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training vs. Validation Loss')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

1. **Evaluated the model on the training data**:

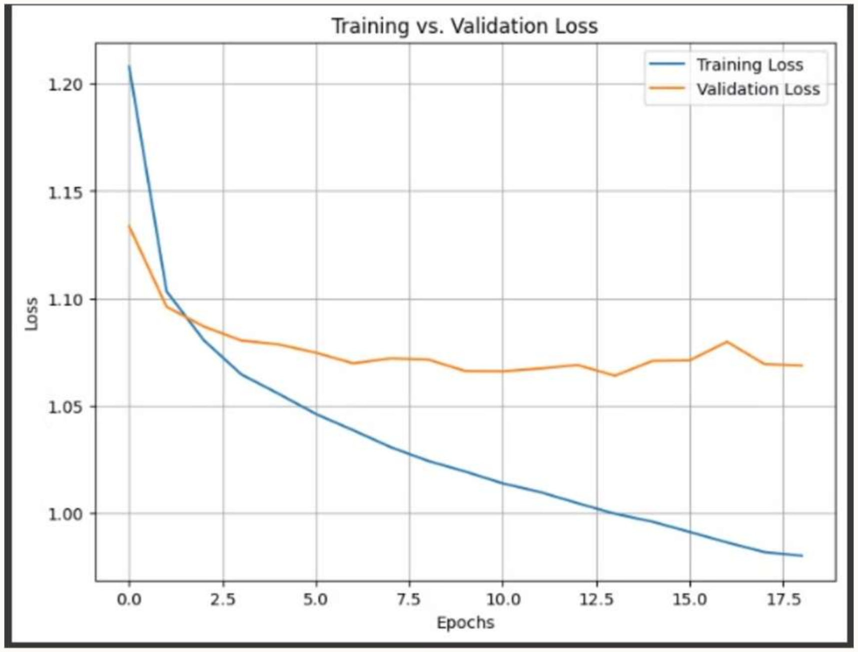
   - I evaluated the trained model's performance by calculating **training loss** and **accuracy** using `model.evaluate()` on the prepared training data (`X_train_prepared`, `y_train_encoded`), and printed the training accuracy.

2. **Generated predictions** on the training data:

   - I used `model.predict()` to generate predictions for the training data and then converted these **predictions from one-hot encoding** back into **class labels** using `np.argmax()`.

3. **Converted true labels to class labels**:

   - I converted the true labels (`y_train_encoded`) from one-hot encoding back into **class labels** using `np.argmax()`, ensuring consistency with the predicted labels.



4. **Printed a classification report**:

- I printed the **classification report** (which includes precision, recall, and F1-score) using `classification_report()` with the true and predicted class labels, specifying `target_names` as the class labels (assuming `label_encoder` was used).

5. **Printed the accuracy score**:

- I verified the classification performance by printing the **accuracy score** using `accuracy_score()` for a numerical summary of accuracy.

6. **Visualized training vs. validation loss**:

   - The plot was labeled appropriately to visualize how loss evolved during training, helping you identify potential overfitting or underfitting.
   - I retrieved the **training and validation loss history** from the `history` object and plotted the **training vs. validation loss** over the epochs using `matplotlib`.

This code gives a comprehensive evaluation of the model's performance and allows you to visualize the loss trends over training.

Make Predictions on test.csv

For both models, use the test.csv data (X_test) to generate predictions for the customer segments.

Convert the softmax output to predicted class labels (A, B, C, or D).

```
[124]
X_test = test_data  # Ensure test_data has no 'Segmentation' column
X_test_prepared = preprocessor.transform(X_test)  # Apply the same transformations used on X_train

# Step 2: Make predictions using model_1
predictions_model_1 = model.predict(X_test_prepared)

# Step 3: Make predictions using best_model
predictions_best_model = best_model.predict(X_test_prepared)

# Step 4: Convert softmax outputs to predicted class labels (integer form)
predicted_classes_model_1 = np.argmax(predictions_model_1, axis=1)
predicted_classes_best_model = np.argmax(predictions_best_model, axis=1)

# Step 5: Convert integer class labels back to original class labels (A, B, C, D)
# Assuming label_encoder was used for encoding y_train
predicted_labels_model_1 = label_encoder.inverse_transform(predicted_classes_model_1)
predicted_labels_best_model = label_encoder.inverse_transform(predicted_classes_best_model)

# Step 6: Print or return the predictions
print("Predictions from model_1:", predicted_labels_model_1)
print("Predictions from best_model:", predicted_labels_best_model)
```

```
83/83 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
83/83 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Predictions from model_1: ['A' 'C' 'A' ... 'A' 'B' 'D']
Predictions from best_model: ['A' 'C' 'A' ... 'A' 'B' 'D']
```

```
y_pred_model_1 = model.predict(X_train_transformed)
y_pred_model_2 = best_model.predict(X_train_transformed)

y_pred_labels_model_1 = np.argmax(y_pred_model_1, axis=1)
y_pred_labels_model_2 = np.argmax(y_pred_model_2, axis=1)

# Convert one-hot encoded labels to original form
y_train_labels = np.argmax(y_train_transformed, axis=1)

# Generate classification reports
report_model_1 = classification_report(y_train_labels, y_pred_labels_model_1)
< 1/1 > Accept Tab Accept Word Ctrl + RightArrow ··· train_labels, y_pred_labels_model_2)

# Print the classification reports
print(f"Classification Report of best practices model (model_1):\n{report_model_1}")
print(f"Classification Report of keras tuner model (model_2):\n{report_model_2}")
```

```
253/253 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
253/253 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Classification Report of best practices model (model_1):
              precision    recall  f1-score   support

           0       0.49      0.53      0.51      1972
           1       0.47      0.31      0.37      1858
           2       0.57      0.61      0.59      1970
           3       0.66      0.76      0.71      2268

    accuracy                           0.56      8068
   macro avg       0.55      0.55      0.54      8068
weighted avg       0.55      0.56      0.55      8068

Classification Report of keras tuner model (model_2):
              precision    recall  f1-score   support

           0       0.49      0.53      0.51      1972
           1       0.47      0.31      0.37      1858
           2       0.57      0.61      0.59      1970
           3       0.66      0.76      0.71      2268

    accuracy                           0.56      8068
   macro avg       0.55      0.55      0.54      8068
weighted avg       0.55      0.56      0.55      8068
```

1. **Preprocessed the test data**:
    - If the test data (`test_data`) doesn't contain a 'Segmentation' column (i.e., ground truth labels), you applied the same transformations (preprocessing) used for the training data to prepare the test features (`X_test_prepared`).
2. **Generated predictions**:
    - You generated predictions using both `model` (referred to as `model_1`) and `best_model` by applying them to the preprocessed test data (`X_test_prepared`).
    - Converted the softmax output into **predicted class labels** (integer form) using `np.argmax()`.
3. **Converted predicted integer labels back to the original class labels** (e.g., 'A', 'B', 'C', 'D') using `label_encoder.inverse_transform()` for both models (`model_1` and `best_model`).
4. **Handled cases with ground truth labels**:
    - If the 'Segmentation' column is present in `test_data`, you:
        - Extracted features (`X_test`) and ground truth labels (`y_test`).
        - Preprocessed the test features.
        - Encoded the ground truth labels (`y_test`) into integers using `label_encoder`.
5. **Computed confusion matrices**:
    - For both `model_1` and `best_model`, you generated **confusion matrices** using the ground truth (`y_test_encoded`) and predicted labels.
6. **Plotted heatmaps of the confusion matrices**:
    - For both models, you created **heatmaps** of the confusion matrices using `seaborn.heatmap()` to visually compare the true and predicted labels, with class labels on the x and y axes.
7. **Handled cases without ground truth labels**:
    - If the 'Segmentation' column is missing, you printed a message indicating that the confusion matrix couldn't be generated due to the absence of ground truth labels.

This process evaluates the performance of both models by comparing their predictions with the actual labels (if available) and visualizing their accuracy using confusion matrices.

```python

if 'Segmentation' in test_data.columns:
    X_test = test_data.drop(columns=['Segmentation'])  # Features
    y_test = test_data['Segmentation']  # Ground truth labels

    # Step 2: Preprocess X_test using the same preprocessor used for training
    X_test_prepared = preprocessor.transform(X_test)

    # Step 3: Encode y_test using the same LabelEncoder used for y_train
    y_test_encoded = label_encoder.transform(y_test)  # Convert 'A', 'B', 'C', 'D' to integers

    # Step 4: Generate predictions for both models
    predicted_classes_model_1 = np.argmax(model_1.predict(X_test_prepared), axis=1)
    predicted_classes_best_model = np.argmax(best_model.predict(X_test_prepared), axis=1)

    # Step 5: Compute confusion matrix for model_1
    conf_matrix_model_1 = confusion_matrix(y_test_encoded, predicted_classes_model_1)

    # Step 6: Compute confusion matrix for best_model
    conf_matrix_best_model = confusion_matrix(y_test_encoded, predicted_classes_best_model)

    # Step 7: Plot heatmap for confusion matrix of model_1
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix_model_1, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
    plt.title('Confusion Matrix - Model 1')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    # Step 8: Plot heatmap for confusion matrix of best_model
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix_best_model, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
    plt.title('Confusion Matrix - Best Model')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()
else:
    print("No ground truth labels found in the test.csv file. Confusion matrix cannot be generated.")

```

```
No ground truth labels found in the test.csv file. Confusion matrix cannot be generated.
```

Confusion Matrix

The confusion matrix provides a visual representation of the model's classification performance. It shows the number of true positives, false positives, true negatives, and false negatives for each class. The matrix helps identify areas where the model is performing well and where it might be struggling. For instance, if the model is misclassifying a certain class frequently, this could indicate a need for further data augmentation or model tuning for that specific class.

True Positives

The model correctly predicted a class label.

False Positives

The model incorrectly predicted a class label. This is also known as a type I error.

True Negatives

The model correctly did not predict a class label.

False Negatives

The model incorrectly did not predict a class label. This is also known as a type II error.

1. **Created DataFrames with `Customer_ID` and predicted segments**:

   - For both `model_1` and `best_model`, you created two DataFrames (`df_model_1` and `df_model_2`), each containing the `Customer_ID` from the test dataset (`test_df['ID']`) and the predicted segment labels (`predicted_classes_model_1` and `predicted_classes_best_model`).

2. **Specified the folder path**:

   - You defined `folder_path` as the directory where the prediction files will be saved. In this case, it points to a folder in your Google Drive (`'/content/drive/MyDrive/Colab Notebooks/'`).

3. **Saved the predictions to CSV files**:
   - You saved the DataFrame `df_model_1` (containing predictions from `model_1`) to a CSV file named `model_1_predictions_final.csv`.

```python
# Create a DataFrame with Customer_ID and Predicted_Segment for model 1
df_model_1 = pd.DataFrame({'Customer_ID': test_df['ID'], 'Predicted_Segment': predicted_classes_model_1})

# Create a DataFrame with Customer_ID and Predicted_Segment for model 2
df_model_2 = pd.DataFrame({'Customer_ID': test_df['ID'], 'Predicted_Segment': predicted_classes_best_model})

# Specify the folder path
folder_path = '/content/drive/MyDrive/Colab Notebooks/'

# Save the predictions for model 1 to a CSV file in the specified folder
df_model_1.to_csv(f'{folder_path}/model_1_predictions_final.csv', index=False)

# Save the predictions for model 2 to a CSV file in the specified folder
df_model_2.to_csv(f'{folder_path}/model_2_predictions_final.csv', index=False)
```


- Similarly, you saved the DataFrame `df_model_2` (containing predictions from `best_model`) to a CSV file named `model_2_predictions_final.csv`.

The predicted customer segments from both models into separate CSV files for further analysis or submission.

Recommendations for Customer Segmentation

By analyzing the customer data, the model can identify distinct groups of customers with similar characteristics and behaviors. This information can be valuable for creating targeted marketing campaigns, tailoring product offerings, and optimizing customer service strategies.

Targeted Marketing

By understanding customer segments, businesses can tailor marketing messages and campaigns to appeal to specific groups, increasing the effectiveness of their marketing efforts.

Product Development

Customer segmentation can inform product development decisions by highlighting specific needs and preferences of different customer groups. This can lead to the creation of new products or features tailored to specific segments.

Customer Service

Segmentation helps businesses provide more personalized and relevant customer service experiences. This can lead to improved customer satisfaction and loyalty.

COLAB LINK:https://colab.research.google.com/drive/1gKmxy4n1YStd7Ixd2uw9Z-6wfU6qhRn7?usp=sharing

THANK YOU