Let's go through the code step-by-step, focusing specifically on the input portions. I’ll ignore the output steps, as you requested.

1. \*\*Library Imports\*\*

```python

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

import tensorflow as tf

```

These lines import libraries for data handling, visualization, and machine learning. `pandas` and `numpy` help with data manipulation, `seaborn` and `matplotlib` are for visualization, and `tensorflow` is for building neural networks.

2. \*\*Loading the Dataset\*\*

```python

df = pd.read\_csv('Churn\_Modelling.csv')

df.head()

```

Here, `pd.read\_csv` loads the CSV file named `'Churn\_Modelling.csv'` into a `DataFrame` called `df`. The line `df.head()` displays the first few rows, giving a quick look at the data.

3. \*\*Data Summary\*\*

```python

df.info()

```

The `df.info()` function provides an overview of the dataset’s structure, including column names, data types, and non-null counts.

4. \*\*Visualizing Target Column Distribution\*\*

```python

plt.xlabel('Exited')

plt.ylabel('Count')

df['Exited'].value\_counts().plot.bar()

plt.show()

```

These lines create a bar plot to visualize the distribution of the target column, `Exited`, showing the count of customers who exited versus those who did not.

5. \*\*Checking Counts of a Categorical Column\*\*

```python

df['Geography'].value\_counts()

```

Here, `df['Geography'].value\_counts()` counts how often each unique value appears in the `Geography` column, which shows the distribution across different regions (like France, Germany, and Spain).

6. \*\*Encoding Categorical Variables (Geography and Gender)\*\*

```python

df = pd.concat([df, pd.get\_dummies(df['Geography'], prefix='Geo')], axis=1)

df = pd.concat([df, pd.get\_dummies(df['Gender'])], axis=1)

df.info()

```

These lines use `pd.get\_dummies()` to convert categorical variables into numeric dummy variables for machine learning. `Geography` values are split into new columns `Geo\_France`, `Geo\_Germany`, and `Geo\_Spain`, while `Gender` splits into `Female` and `Male`. `pd.concat` then merges these dummy columns back into the original DataFrame.

7. \*\*Dropping Unnecessary Columns\*\*

```python

df.drop(columns=['RowNumber', 'CustomerId', 'Surname', 'Geography', 'Gender'], inplace=True)

df.head()

```

This line removes columns that are not useful for model training, such as IDs or redundant columns. `inplace=True` ensures the changes are made directly to `df`.

8. \*\*Preparing Target and Features\*\*

```python

y = df['Exited'].values

x = df.loc[:, df.columns != 'Exited'].values

```

Here, the target variable `y` is set to the values in the `Exited` column, while `x` contains all remaining columns except `Exited`. This splits the data into the features (`x`) and target (`y`) for machine learning.

9. \*\*Splitting Data into Training and Testing Sets\*\*

```python

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, random\_state=20, test\_size=0.25)

```

The `train\_test\_split` function divides the data into training and testing sets. `random\_state=20` ensures reproducibility, and `test\_size=0.25` specifies that 25% of the data goes into the test set, while 75% goes into the training set.

10. \*\*Scaling the Features\*\*

```python

from sklearn.preprocessing import StandardScaler

std\_x = StandardScaler()

x\_train = std\_x.fit\_transform(x\_train)

x\_test = std\_x.transform(x\_test)

```

Scaling normalizes the feature values to improve model performance. `StandardScaler()` transforms `x\_train` and `x\_test` so each feature has a mean of 0 and a standard deviation of 1.

Here is the code explained step-by-step, focusing specifically on the input portions.

1. \*\*Checking the Shape of the Training Data\*\*

```python

x\_train.shape

```

This line returns the shape of `x\_train`, which is the training data containing feature values. Here, `(7500, 13)` indicates 7,500 rows and 13 columns (features) in the training dataset.

2. \*\*Importing TensorFlow for Model Building\*\*

```python

import tensorflow as tf

from tensorflow.keras.layers import Dense, Conv1D, Flatten

from tensorflow.keras.models import Sequential, Model

```

This code imports the necessary TensorFlow components. `Dense`, `Conv1D`, and `Flatten` are layers commonly used in neural networks, while `Sequential` allows for building the model in a sequential (stacked) manner.

3. \*\*Building the Model\*\*

```python

model = Sequential()

model.add(Flatten(input\_shape=(13,)))

model.add(Dense(100, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

```

- `model = Sequential()` initializes a new sequential model.

- `model.add(Flatten(input\_shape=(13,)))` flattens the input data (13 features) into a 1D array. This layer ensures that data is in a compatible shape for dense layers.

- `model.add(Dense(100, activation='relu'))` adds a dense layer with 100 neurons and ReLU activation, introducing non-linearity to learn complex patterns.

- `model.add(Dense(1, activation='sigmoid'))` adds the output layer with 1 neuron and a sigmoid activation, which outputs a probability between 0 and 1 for binary classification.

4. \*\*Compiling the Model\*\*

```python

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

```

This line compiles the model. The parameters:

- `optimizer='adam'` specifies the Adam optimizer for efficient training.

- `metrics=['accuracy']` tracks the accuracy during training.

- `loss='BinaryCrossentropy'` uses binary cross-entropy, appropriate for binary classification tasks.

5. \*\*Training the Model\*\*

```python

model.fit(x\_train, y\_train, batch\_size=64, validation\_split=0.1, epochs=100)

```

- `model.fit(x\_train, y\_train, batch\_size=64, validation\_split=0.1, epochs=100)` starts training the model using the training data (`x\_train` and `y\_train`).

- `batch\_size=64` divides the dataset into batches of 64 samples, updating model parameters after each batch.

- `validation\_split=0.1` reserves 10% of the training data for validation to monitor overfitting.

- `epochs=100` trains the model for 100 iterations over the entire training dataset.

Certainly! Let's break down the code line by line, explaining each part in detail. Here’s the code snippet you provided, organized for clarity:

```python

pred = model.predict(x\_test)

```

### 1. Model Prediction

- \*\*Line\*\*: `pred = model.predict(x\_test)`

- \*\*Explanation\*\*: This line calls the `predict` method on the `model` object (which is presumably a trained machine learning model, like a logistic regression or neural network). The method takes `x\_test` as an input, which is the feature set for the test dataset. The model will output predictions for each instance in the test set, and these predictions are stored in the `pred` variable.

```plaintext

79/79 [==============================] - 0s 567us/step

```

### 2. Progress Display

- \*\*Line\*\*: (This is an output message, not code.)

- \*\*Explanation\*\*: This output indicates that the model has processed 79 batches of data (which corresponds to 79 samples in this context) during the prediction phase. The message also shows that it took approximately 567 microseconds per step to complete the predictions.

```python

y\_pred = []

```

### 3. Initialize Predictions List

- \*\*Line\*\*: `y\_pred = []`

- \*\*Explanation\*\*: This line initializes an empty list named `y\_pred`. This list will be used to store the final predicted class labels (0 or 1) for each instance in the test set after applying a threshold to the model’s predictions.

```python

for val in pred:

if val > 0.5:

y\_pred.append(1)

else:

y\_pred.append(0)

```

### 4. Thresholding Predictions

- \*\*Lines\*\*:

- `for val in pred:`

- `if val > 0.5:`

- `y\_pred.append(1)`

- `else:`

- `y\_pred.append(0)`

- \*\*Explanation\*\*:

- \*\*Loop\*\*: This loop iterates over each predicted value (`val`) in the `pred` array. Each value represents the predicted probability that a given instance belongs to the positive class (label 1).

- \*\*Thresholding\*\*: The conditional statement checks if the predicted probability (`val`) is greater than 0.5:

- If true, it appends `1` to the `y\_pred` list, indicating a prediction of the positive class.

- If false, it appends `0`, indicating a prediction of the negative class.

- This method of thresholding converts continuous probability outputs into binary class labels.

```python

from sklearn.metrics import accuracy\_score, confusion\_matrix, ConfusionMatrixDisplay

```

### 5. Import Evaluation Metrics

- \*\*Line\*\*: `from sklearn.metrics import accuracy\_score, confusion\_matrix, ConfusionMatrixDisplay`

- \*\*Explanation\*\*: This line imports specific functions from the `sklearn.metrics` module:

- `accuracy\_score`: To calculate the accuracy of the model’s predictions (the proportion of correct predictions).

- `confusion\_matrix`: To compute the confusion matrix, which summarizes the performance of the classification model.

- `ConfusionMatrixDisplay`: To visualize the confusion matrix.

```python

accuracy\_score(y\_test, y\_pred)

```

### 6. Calculate Accuracy

- \*\*Line\*\*: `accuracy\_score(y\_test, y\_pred)`

- \*\*Explanation\*\*: This line computes the accuracy of the model by comparing the true labels (`y\_test`) with the predicted labels (`y\_pred`). It returns a floating-point number between 0 and 1, representing the proportion of correct predictions.

```python

cm = confusion\_matrix(y\_test, y\_pred)

```

### 7. Compute Confusion Matrix

- \*\*Line\*\*: `cm = confusion\_matrix(y\_test, y\_pred)`

- \*\*Explanation\*\*: This line calculates the confusion matrix using the true labels (`y\_test`) and the predicted labels (`y\_pred`). The confusion matrix is a 2x2 table that shows the number of true positives, true negatives, false positives, and false negatives, which helps in evaluating the performance of the classification model.

```python

display = ConfusionMatrixDisplay(cm)

```

### 8. Prepare to Display Confusion Matrix

- \*\*Line\*\*: `display = ConfusionMatrixDisplay(cm)`

- \*\*Explanation\*\*: This line creates an instance of the `ConfusionMatrixDisplay` class, passing the confusion matrix (`cm`) as an argument. This prepares the confusion matrix for visualization.

```python

display.plot()

```

### 9. Visualize Confusion Matrix

- \*\*Line\*\*: `display.plot()`

- \*\*Explanation\*\*: This line calls the `plot` method on the `display` object to generate and show a visual representation of the confusion matrix. This visualization helps in interpreting the model's performance more intuitively.

---

### Neural Network Classifier

```python

from sklearn.neural\_network import MLPClassifier

```

### 10. Import MLP Classifier

- \*\*Line\*\*: `from sklearn.neural\_network import MLPClassifier`

- \*\*Explanation\*\*: This line imports the `MLPClassifier` class from the `sklearn.neural\_network` module. The `MLPClassifier` is a type of feedforward artificial neural network.

```python

nn\_classifier = MLPClassifier(hidden\_layer\_sizes=(100), activation='logistic', max\_iter=300)

```

### 11. Initialize Neural Network Classifier

- \*\*Line\*\*: `nn\_classifier = MLPClassifier(hidden\_layer\_sizes=(100), activation='logistic', max\_iter=300)`

- \*\*Explanation\*\*: This line initializes an instance of the `MLPClassifier` with the following parameters:

- `hidden\_layer\_sizes=(100)`: Defines the structure of the neural network, specifying one hidden layer with 100 neurons.

- `activation='logistic'`: Sets the activation function for the neurons in the hidden layer to the logistic (sigmoid) function.

- `max\_iter=300`: Specifies the maximum number of iterations (epochs) for training the model.

```plaintext

/home/pratik/.local/lib/python3.8/site-packages/sklearn/neural\_network/\_multilayer\_perceptron.py:702: ConvergenceWarning:

Stochastic Optimizer: Maximum iterations (300) reached and the optimization hasn't converged yet.

warnings.warn(

```

### 12. Convergence Warning

- \*\*Line\*\*: (This is an output message, not code.)

- \*\*Explanation\*\*: This warning indicates that the optimization algorithm used to train the neural network did not converge within the maximum allowed iterations (300). This could imply that the model may not have fully learned the training data, which may affect its performance.

```python

MLPClassifier(activation='logistic', hidden\_layer\_sizes=100, max\_iter=300)

```

### 13. Display MLP Classifier

- \*\*Line\*\*: (This is an output message, not code.)

- \*\*Explanation\*\*: This line displays the configuration of the `MLPClassifier`, indicating that it is set up with the logistic activation function, one hidden layer of 100 neurons, and a maximum of 300 iterations.

```python

nn\_classifier.fit(x\_train, y\_train)

```

### 14. Train Neural Network Classifier

- \*\*Line\*\*: `nn\_classifier.fit(x\_train, y\_train)`

- \*\*Explanation\*\*: This line trains the neural network classifier using the training data (`x\_train`) and the corresponding target labels (`y\_train`). The model learns from the data during this process.

```python

y\_pred2 = nn\_classifier.predict(x\_test)

```

### 15. Predict with Neural Network

- \*\*Line\*\*: `y\_pred2 = nn\_classifier.predict(x\_test)`

- \*\*Explanation\*\*: This line uses the trained neural network classifier to make predictions on the test set (`x\_test`). The predicted labels are stored in `y\_pred2`.

```python

accuracy\_score(y\_pred=y\_pred2, y\_true=y\_test)

```

### 16. Calculate Neural Network Accuracy

- \*\*Line\*\*: `accuracy\_score(y\_pred=y\_pred2, y\_true=y\_test)`

- \*\*Explanation\*\*: This line calculates the accuracy of the neural network model by comparing the true labels (`y\_test`) with the predicted labels (`y\_pred2`). It returns the proportion of correct predictions made by the neural network.

```python

nn\_classifier.score(x\_test, y\_test)

```

### 17. Get Neural Network Score

- \*\*Line\*\*: `nn\_classifier.score(x\_test, y\_test)`

- \*\*Explanation\*\*: This line computes the accuracy score of the neural network model on the test set directly using the `score` method. It combines the prediction and accuracy calculation steps, returning the same proportion of correct predictions.

### Summary

This code provides a comprehensive process for training a logistic regression model, evaluating its performance, training a neural network model, and comparing their accuracies. The evaluation metrics and visualizations (like the confusion matrix) help to assess how well the models perform on unseen data.