ASSIGNMENT - 2

MACHINE LEARNING (BITS F464)

SUBMITTED BY

Harshvardhan Jouhary (2020B2A31623P)

Aryaman Chauhan (2020B5A72006P)

Devansh (2020B5A72001P)

Table of Contents

1. Description

The census-income dataset contains census information for 48,842 people. It has 14 attributes for each person (age, workclass, fnlwgt, education, education-num, marital-status, occupation, relationship, race, sex, capital-gain, capital-loss, hours-per-week, and native-country) and a Boolean attribute class classifying the input of the person as belonging to one of two categories >50K, <=50K. Given the attribute values, the prediction problem here is classifying whether a person's salary is >50K or <= 50K.

Properties of Data

- Number of Instances - 48842 instances, mix of continuous and discrete (train=32561, test=16281)
    - 45222 if instances with unknown values are removed (train=30162, test=15060)
    - Number of Attributes: 6 continuous, 8 nominal attributes

- Attribute Information:

1) age: continuous
2) workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
3) fnlwgt: continuous
4) education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st4th, 10th, Doctorate, 5th-6th, Preschool
5) education-num: continuous
6) marital-status: Marriedciv-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AFspouse
7) occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Profspecialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
8) relationship: Wife, Own-child, Husband, Not-infamily, Other-relative, Unmarried
9) race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
10) sex: Female, Male
11) capital-gain: continuous
12) capital-loss: continuous
13) hours-perweek: continuous
14) native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, DominicanRepublic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands
15) class: >50K, <=50K To get started with the model, run environment.yml file to initialize the conda environment with relevant libraries.

2. Pre-Processing of Training Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
import torch
import torch.nn as nn
import seaborn as sns
import json
import pickle
import math
%matplotlib inline
```

The libraries of pandas, numpy, and matplotlib.pyplot are imported. Also, the regression algorithms about to be used are imported from the scikit-learn package. The training and testing data are loaded into the data frames train_df and test_df using the Panda library. Initially, given data contains " ?" to be considered null. Just to make sure those are null values, all the question marks(" ?") were replaced with NULL.

2.1 Handling Missing Values

```
def dfFillNaN(data_df):
    median_values = data_df.median(numeric_only=True)
    data_df = data_df.fillna(median_values)
    processed_df = pd.DataFrame()
```

All the NULL (or NaN) values of the floating point type in a column are replaced with the median value of that column. Here, there are seven attribute columns with float type. The median values of these columns are listed above. We should use the median of a column to fill in the missing data because the median preserves the central tendency and is not significantly affected by outliers, unlike the mode.

```
    for column in data_df.columns:
        if data_df[column].dtype == 'object':
            mode_value = data_df[column].mode()[0]
            processed_df[column] = data_df[column].fillna(mode_value)
        else:
            processed_df[column] = data_df[column]
    return processed_df
```

Now, all the NULL values of the object(or String) data type are replaced with the mode of respective columns. The mode or most occurring values of these columns are listed above. The 'Inplace = True' parameter lets us modify the original objects directly without creating a separate, new object.

This concludes the handling or replacement of missing data in the training data set.

2.2 Label Mapping

We will encode our data to ensure it works for classifiers that don't take non-numeric values. We've ensured that this encoding is meaningful like education increases from preschool to doctorate. The new Encoding can be found here.

```
  "workclass": {
    "State-gov": 6,    "Self-emp-not-inc": 2,    "Private": 4,    "Federal-gov": 7,    "Local-gov": 5,    "Self-emp-inc": 3,
    "Without-pay": 1,    "Never-worked": 0
  },
  "class": {
      "<=50K": 0,    ">50K": 1
  },
  "native-country": {
    "United-States": 38,    "Cuba": 4,    "Jamaica": 22,    "India": 18,    "Mexico": 25,    "South": 34,  Puerto-Rico": 32,
    "Honduras": 15,    "England": 8,    "Canada": 1,    "Germany": 10,    "Iran": 19,    "Philippines": 29,
    "Italy": 21,    "Poland": 30,    "Columbia": 3,    "Cambodia": 0,    "Thailand": 36,    "Ecuador": 6,    "Laos": 24,
    "Taiwan": 35,    "Haiti": 13,    "Portugal": 31,    "Dominican-Republic": 5,    "El-Salvador": 7,    "France": 9,
    "Guatemala": 12,  "China": 2,    "Japan": 23,    "Yugoslavia": 40,    "Peru": 28,  "Outlying-US(Guam-USVI-etc)": 27,
    "Scotland": 33,    "Trinadad&Tobago": 37,    "Greece": 11,    "Nicaragua": 26,    "Vietnam": 39,    "Hong": 16,
    "Ireland": 20,    "Hungary": 17,    "Holand-Netherlands": 14
  },
  "sex": {
    "Male": 1,    "Female": 0
  },
  "race": {
    "White": 4,    "Black": 2,    "Asian-Pac-Islander": 3,    "Amer-Indian-Eskimo": 1,    "Other": 0
  },
  "relationship": {
    "Not-in-family": 1,    "Husband": 4,    "Wife": 2,    "Own-child": 5,    "Unmarried": 3,    "Other-relative": 0
  },
  "occupation": {
    "Adm-clerical": 3,    "Exec-managerial": 12,    "Handlers-cleaners": 2,    "Prof-specialty": 13,
"Other-service": 4,    "Sales": 9,    "Craft-repair": 8,    "Transport-moving": 7,    "Farming-fishing": 6,
"Machine-op-inspct": 5,    "Tech-support": 10,    "Protective-serv": 11,    "Armed-Forces": 0,    "Priv-house-serv": 1
  },
  "marital-status": {
    "Never-married": 4,    "Married-civ-spouse": 2,    "Divorced": 0,    "Married-spouse-absent": 3,
    "Separated": 5,    "Married-AF-spouse": 1,    "Widowed": 6
  },
  "education": {
    "Bachelors": 12,    "HS-grad": 13,    "11th": 6,    "Masters": 14,    "9th": 4,    "Some-college": 11,
"Assoc-acdm": 10,    "Assoc-voc": 9,    "7th-8th": 3,    "Doctorate": 15,    "Prof-school": 8,    "5th-6th": 2,
    "10th": 5,    "1st-4th": 1,    "Preschool": 0,    "12th": 7
  })
```

```python
def dfEncoder(data_df, label_mapping_file="label_mapping.json"):
    # Load the label encoding mapping from the JSON file
    with open(label_mapping_file, 'r') as mapping_file:
        label_mapping = json.load(mapping_file)

    # Create a new DataFrame to store the encoded data
    encoded_data_df = pd.DataFrame()

    for column in data_df.columns:
        if column not in label_mapping:
            encoded_data_df[column] = data_df[column]

    # Iterate over the columns of the input DataFrame
    for column in data_df.columns:
        if column in label_mapping:
            # Create a label encoder and set the classes using the label mapping
            label_encoder = LabelEncoder()
            label_encoder.classes_ = np.append(list(label_mapping[column].values

            # Apply the label encoding to the selected column
            encoded_column = data_df[column].astype(str).fillna(data_df[column].
            encoded_data_df[column+'_n'] = encoded_column
            # encoded_data_df[column] = data_df[column] #Was made to test if enc

    return encoded_data_df
```

```python
train_df = dfEncoder(dfFillNaN(train_df))
test_df = dfEncoder(test_df)
train_df.head()
```

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week	workclass_n	education_n	marital-status_n
0	39	77516	13	2174	0	40	6	12	4
1	50	83311	13	0	0	13	2	12	2
2	38	215646	9	0	0	40	4	13	0
3	53	234721	7	0	0	40	4	6	2
4	28	338409	13	0	0	40	4	12	2

3. Training Data

3.1 Scaling Data

We will scale our data to ensure that one column does not affect the data more than another. StandardScaler standardization from the Scikit-Learn package is used. Using this, each attribute value is subtracted from the mean of the attribute and divided by the standard deviation of the attribute.

$$z = \frac{x - \mu}{\sigma}$$

where:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

as mean and standard deviation:

$$\sigma = \sqrt{\sum_{i=1}^{N} \frac{(x_i - \mu)^2}{N}}$$

```
sc = StandardScaler()
X_train = train_df.drop(columns="class_n",axis="columns")
Y_train = train_df.class_n
X_test = test_df.drop(columns="class_n",axis="columns")
Y_test = test_df.class_n
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

3.2 Naive Bayes Classification

The Naive Bayes (NB) classifier is based on the Bayes' Theorem with the 'naive' assumption of feature independence. It calculates the probability of a target class given the observed features. The algorithm calculates the conditional probability of each class given the feature values and assigns the class with the highest probability as the predicted class.

A popular variant of the Naive Bayes Classifier is the Gaussian Naive Bayes which assumes the features follow a Gaussian (Normal) distribution. It is suitable for continuous numerical features. It estimates the mean and standard deviation of each feature for each class and models the likelihood of feature values using a Gaussian distribution to compute the posterior probabilities.

Here, we are using GaussianNB from the Scikit-Learn.

```
naiveGauss = GaussianNB()
X_train_nb, Y_train_nb, X_test_nb, Y_test_nb = X_train.copy(), Y_train.copy(), X_
naiveGauss.fit(X_train_nb, Y_train_nb)
```

```
Y_pred_nb = naiveGauss.predict(X_test_nb)
cm_nb = confusion_matrix(Y_test_nb, Y_pred_nb)
```

```
naiveGauss.score(X_train_nb, Y_train_nb)
```

```
0.8192008844937195
```

```
naiveGauss.score(X_test_nb, Y_test_nb)
```

```
0.8161046618758061
```

As can be seen from the above output, the accuracy is coming out to be 81.9% on training data and 81.6% on testing data.

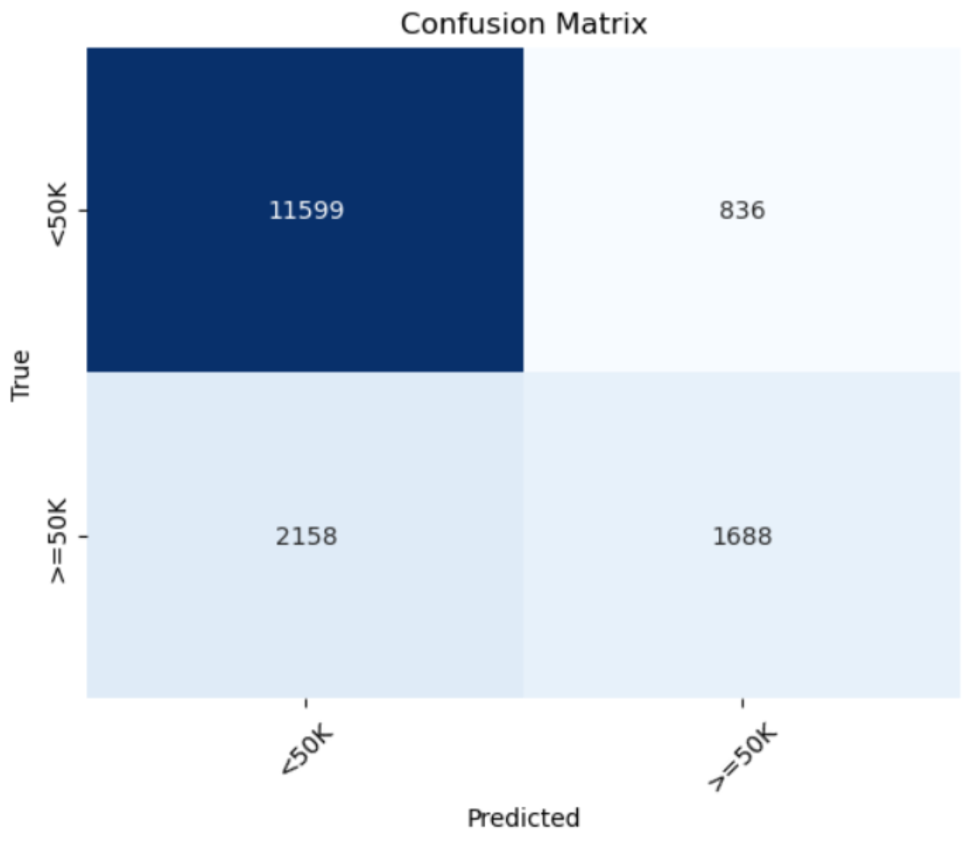
A confusion or error matrix is a table used to evaluate the performance of a classification model. It summarizes the predictions made by a classifier and compares them to the ground truth values. The rows represent the ground truth class labels, and the columns represent the predicted class labels.
True Positive - Instances that are correctly predicted as positive by the model.
False Negative - Instances that are incorrectly predicted as negative by the model.
False Positive - Instances that are incorrectly predicted as positive by the model.
True Negative - Instances that are correctly predicted as negative by the model.

The confusion matrix provides several performance metrics that can be derived from these four values, including

Accuracy: The overall accuracy of the classifier, calculated as (TP + TN) / (TP + FP + FN + TN).
Precision: The ability of the classifier to correctly identify positive instances, calculated as TP / (TP + FP).
Recall: The ability of the classifier to correctly identify all positive instances, calculated as TP / (TP + FN).
Specificity: The ability of the classifier to correctly identify all negative instances, calculated as TN / (TN + FP).
F1 Score: A combined measure of precision and recall, calculated as 2 * (Precision * Recall) / (Precision + Recall).

```
class_labels = ['<50K', '>=50K']
fig_nb, ax_nb = plt.subplots()

# Create a heatmap using seaborn
heatmap = sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax_nt

ax_nb.set_xlabel('Predicted')
ax_nb.set_ylabel('True')
ax_nb.set_title('Confusion Matrix')
ax_nb.xaxis.set_ticklabels(class_labels)
ax_nb.yaxis.set_ticklabels(class_labels)
```

```
(array([0.5, 1.5]), [Text(0.5, 0, '<50K'), Text(1.5, 0, '>=50K')])
```



3.3 Logistic Regression Classifier

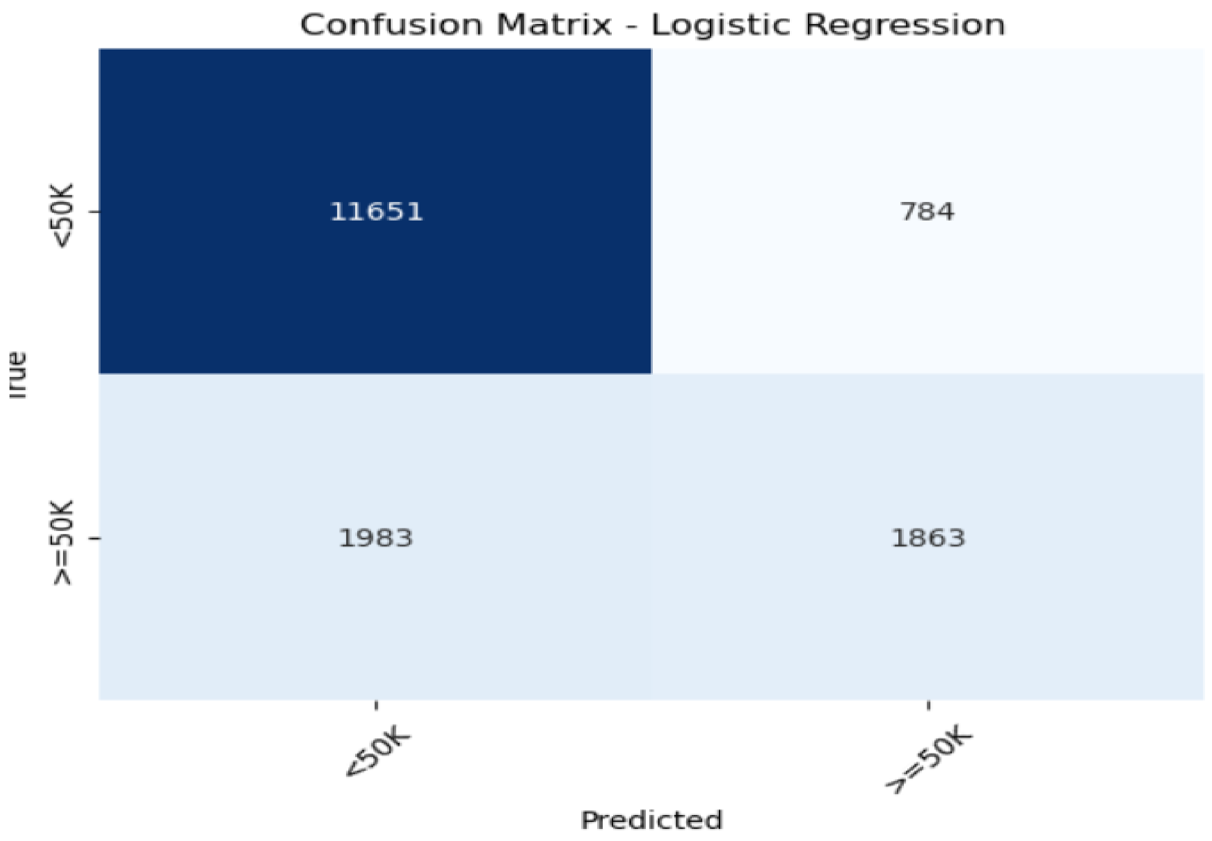
The 'Logistic Regression Classifier' is a binary classification algorithm that predicts the probability of an instance belonging to a specific class. It uses the sigmoid function σ(z), where z represents the linear combination of input features and their corresponding coefficients, to map the linear regression model to a probability value between 0 and 1. By setting a threshold (e.g., 0.5), we can classify instances with predicted probabilities above the threshold as one class and those below as the other.

The optimal values for the coefficients (weights) are estimated by minimizing the cost function and gradient descent as an optimization algorithm to update the coefficients iteratively and gradually. It prevents overfitting and improves generalization by incorporating regularization techniques like Lasso and Ridge, which control the complexity of the model by adding a penalty term to the cost function, discouraging overly large coefficients.

```
logReg = LogisticRegression(penalty=None, random_state=0,max_iter=500, multi_class='ovr',n_jobs=4)
X_train_lr, Y_train_lr, X_test_lr, Y_test_lr = X_train.copy(), Y_train.copy(), X_test.copy(), Y_test.copy()
```

```
Y_pred_lr = logReg.predict(X_test_lr)
cm_lr = confusion_matrix(Y_test_lr, Y_pred_lr)
class_labels = ['<50K', '>=50K']
fig_lr, ax_lr = plt.subplots()

# Create a heatmap using seaborn
heatmap = sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax_lr)

ax_lr.set_xlabel('Predicted')
ax_lr.set_ylabel('True')
ax_lr.set_title('Confusion Matrix - Logistic Regression')
ax_lr.xaxis.set_ticklabels(class_labels)
ax_lr.yaxis.set_ticklabels(class_labels)
```

```
(array([0.5, 1.5]), [Text(0.5, 0, '<50K'), Text(1.5, 0, '>=50K')])
```



```
logReg.score(X_train_lr, Y_train_lr)
```

```
0.8327139829857806
```

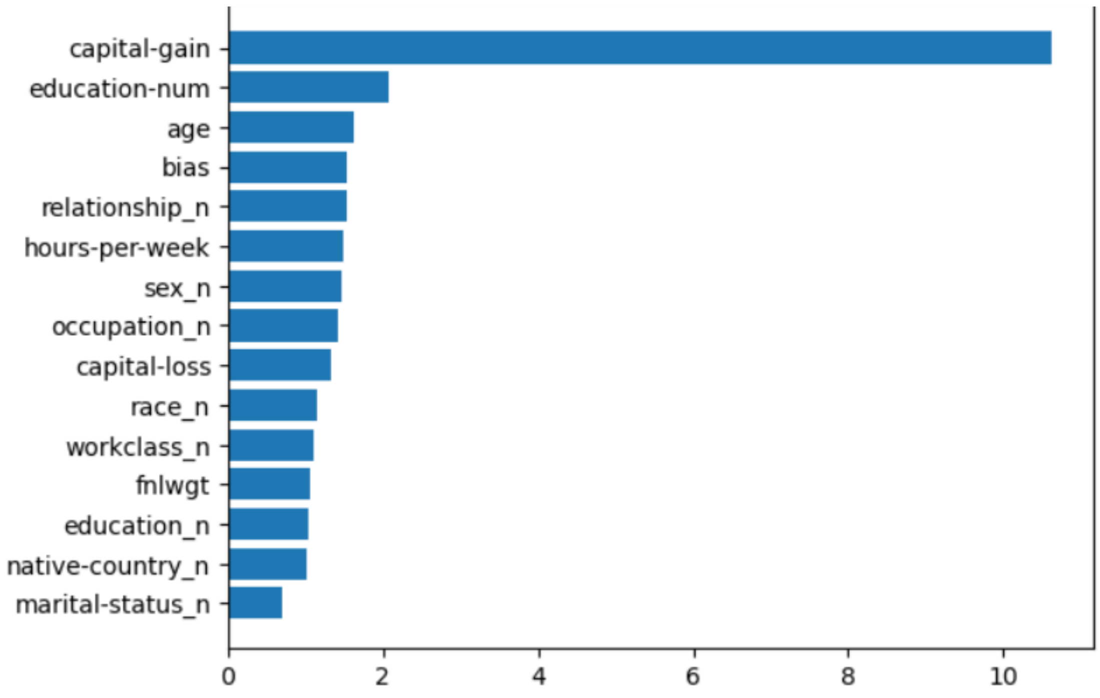
```
logReg.score(X_test_lr, Y_test_lr)
```

```
0.8300472943922363
```

As can be seen from the above output, the accuracy is coming out to be 83.2% on training data and 83.0% on testing data.

The feature importance for Logistic Regression and the equation derived from it are

```
cw = logReg.coef_[0]
cw0 = logReg.intercept_[0]
feature_importance_lr = pd.DataFrame(train_df.drop(columns='class_n', axis='columns').columns, columns = ["attribute"])
feature_importance_lr["importance"] = pow(math.e, cw)

bias_row = pd.DataFrame({'attribute':['bias'],
                         'importance':[abs(cw0)]}, index=[0])
feature_importance_lr = pd.concat([feature_importance_lr, bias_row], ignore_index=True)
feature_importance_lr = feature_importance_lr.sort_values(by = ["importance"], ascending=False)

fig_lrfi, ax_lrfi = plt.subplots()
ax_lrfi.barh(feature_importance_lr.attribute, feature_importance_lr.importance)
```



$$y = 0.00591408x_{14} + 0.37565091x_{13} + 0.13533394x_{12} + 0.42481656x_{11} + 0.3455405x_{10} - 0.3532019x_9 + 0.02593252x_8$$

$$+ 0.10430457x_7 + 0.40112813x_6 + 0.27644223x_5 + 2.36383934x_4 + 0.73036156x_3 + 0.0653556x_2 + 0.48746454x - 1.540632545$$

where x represents as follows: 14:age, 13:fnlwgt, 12:education-num, 11:capital-gain, 10:capital-loss, 9:hours-per-week, 8:workclass_n, 7:education_n, 6:marital-status_n, 5:occupation_n, 4:relationship_n, 3:race_n, 2:sex_n, 1:native-country_n, 0:bias

3.4 Neural Network

The data frames X_train, X_test, Y_train, Y_test are currently in NumPy arrays. They must be converted to PyTorch tensors to utilize the capabilities of the PyTorch library for building and training neural networks. PyTorch tensors enable efficient computation of gradients for backpropagation during neural network training, GPU acceleration for faster training and inference, and easier integration with the PyTorch ecosystem.

```
X_train_nn1 = torch.from_numpy(X_train_nn1).type(torch.float32)
Y_train_nn1 = torch.from_numpy(Y_train_nn1.to_numpy()).type(torch.float32)
X_test_nn1 = torch.from_numpy(X_test_nn1).type(torch.float32)
Y_test_nn1 = torch.from_numpy(Y_test_nn1.to_numpy()).type(torch.float32)
X_train_nn1[:5],Y_train_nn1[:5]
```

Here, the arrays are converted from NumPy to Torch tensors of float32 data type to ensure consistency and compatibility with the model’s requirements.

```
X_test_nn1, X_valid_nn1, Y_test_nn1, Y_valid_nn1 = train_test_split(X_test_nn1, Y_test_nn1, train_size=0.5, random_state=10
X_test_nn1.shape, X_valid_nn1.shape
```

```
(torch.Size([8140, 14]), torch.Size([8141, 14]))
```

The testing tensor is split by 50% into testing and validation tensors.

3.4.1 With 1 Hidden Layer

```
model_nn1 = nn.Sequential(nn.Linear(in_features=14, out_features=10),#Input Layer to Hidden Layer
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=10, out_features=1))#Hidden Layer to Output Layer
```

```
loss_nn1 = nn.BCEWithLogitsLoss()#Non-Linear Function to get probability of class inclusive here
grad_nn1 = torch.optim.SGD(model_nn1.parameters(), lr=0.001)
```

```
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item() # torch.eq() calculates where two tensors are equal
    acc = (correct / len(y_pred)) * 100
    return acc
```

The Neural Network Model using PyTorch's 'nn.Sequential' module is trained. The Input Layer is defined using 'nn. Linear' layer, which expects input with 14 features, i.e., the dataset's attribute columns, produces output with 10 features in the Hidden Layer. The 'nn. The linear' layer performs a linear transformation on the input data by applying weights and biases. The hidden or Non-Linear Activation Function (nn.Sigmoid) layer is used between the input and the output layers. This function squashes the output values between 0 and 1, allowing the model to predict probabilities or make binary classifications. The Output Layer takes the outputs from the previous layer (10 features) and produces a single output by applying linear transformations to map the intermediate features to the final output.

The Loss Function ('nn.BCEWithLogitsLoss()') is used for the classification. It combines the sigmoid activation function and the binary cross-entropy loss into a single function. This function suits models with a single value output as in our dataset. The Gradient Descent Optimizer ('torch.optim.SGD()') performs gradient descent optimization on the model's parameters, i.e., weights and biases, taken as input using 'model_nn1.parameters( )'. The learning rate 'lr' determines the step size at each iteration of the optimization process. It controls the magnitude of the optimizer's adjustment of the model's parameters.

We define a function to calculate the accuracy of the predicted tensor values compared to the true tensor values using 'torch.eq( )', which returns 'True' for equality, else 'False'. We calculate the total number of equal elements, retrieve the scalar value from the resulting tensor, and store this in the 'correct' variable. Dividing this variable by the total number of predictions 'len(y_pred)' gives the model's accuracy.

```
torch.manual_seed(6)
train_loss_values = []
test_loss_values = []
epoch_count = []
train_accuracy_values = []
valid_accuracy_values = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn1.train()

    # Forward Pass
    y_pred_nn1logits = model_nn1(X_train_nn1).squeeze()
    y_pred_nn1 = torch.round(torch.sigmoid(y_pred_nn1logits))

    # Calculating Loss
    losstrain = loss_nn1(y_pred_nn1logits, Y_train_nn1)

    epoch_count.append(epoch)
    train_loss_values.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn1, y_pred=y_pred_nn1)
    train_accuracy_values.append(train_acc)

    # Zero Grad of optimizer
    grad_nn1.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn1.step()
```

```
## Validation
# Setting evaluation mode
model_nn1.eval()

with torch.inference_mode():
    valid_predlogits = model_nn1(X_valid_nn1).squeeze()
    valid_pred = torch.round(torch.sigmoid(valid_predlogits))
    loss_valid = loss_nn1(valid_predlogits, Y_valid_nn1)
    test_loss_values.append(loss_valid.item())
    valid_acc = accuracy_fn(y_true=Y_valid_nn1, y_pred=valid_pred)
    valid_accuracy_values.append(valid_acc)

if epoch%500==0:
    print(f"Epoch: {epoch} | Loss: {losstrain.item()}, Accuracy: {train_acc:.2f}% | Test Loss: {loss_valid.item()}, Test
```

```
model_nn1.eval()
with torch.inference_mode():
    train_pred = model_nn1(X_train_nn1).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn1, y_pred=train_pred)
train_acc
```

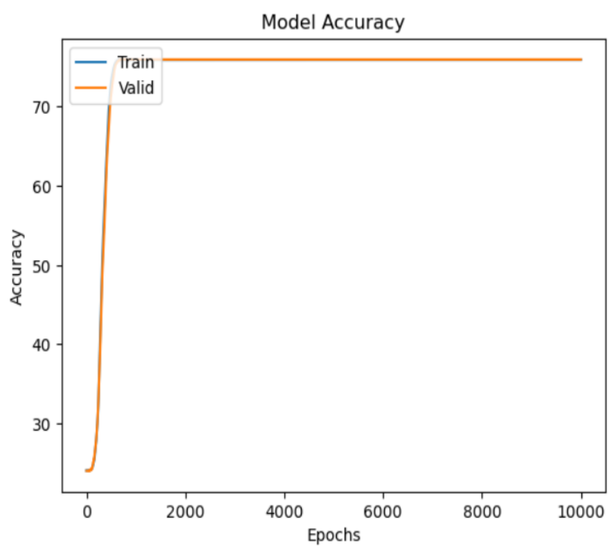
```
75.91904425539757
```

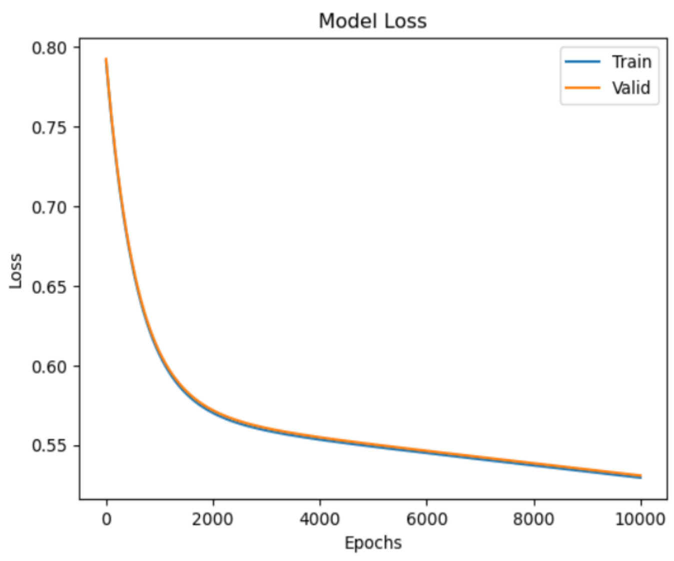
```
model_nn1.eval()
with torch.inference_mode():
    test_pred = model_nn1(X_test_nn1).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn1, y_pred=test_pred)
test_acc
```

```
76.8058968058968
```

The neural network model is trained. The reproducibility of the results is ensured by setting up the random seed using ‘manual_seed(6)’. An epoch refers to a single iteration through the entire training dataset during the training phase of the model. The training loop iterates over 10,000 epochs. ‘Y_pred_nn1logits’ calculates the predicted logits of the model by passing the training data ‘X_train_nn1’. ‘Y_pred_nn1’ computes the predicted labels by applying the sigmoid activation function to the logits and rounding the values to 0 or 1. ‘Losstrain’ computes the loss between the predicted logits and ground truth ‘Y_train_nn1’ using the defined loss function ‘loss_nn1’. ‘Train_acc’ calculates the training accuracy by calling the defined ‘accuracy_fn’ function. ‘grad_nn1.zero_grad( )’ resets the gradients of the optimizer. ‘Losstrain.backward( )’ performs backpropagation to compute the gradients of the loss concerning the model’s parameters. Lastly, ‘grad_nn1.step( )’ updates the model’s parameters using the gradients computed during backpropagation.

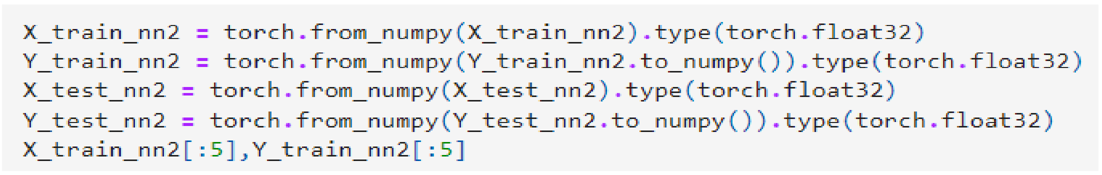
All the above steps are repeated for the validation dataset, and loss and accuracy values of training and validation are stored. The accuracy is calculated and rounded to the nearest integer using ‘torch.round( )’ function. The accuracy on the training dataset is 75.9% and 76.8% on the testing dataset.



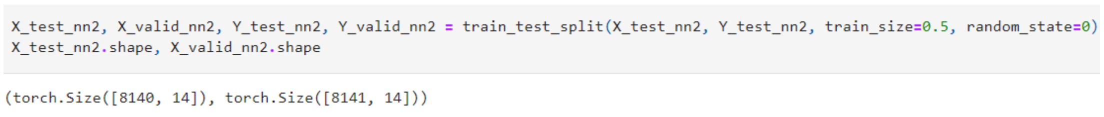


3.4.2 With 2 Hidden Layers

Similar steps as in the case with 1 Hidden layer are repeated. Here, the Input layer has 14 input features and 9 output features that become the input of Hidden Layer 1. This layer also has 9 output features that become the input of Hidden Layer 2. The 9 output features are input for the Output Layer that gives the final output feature.

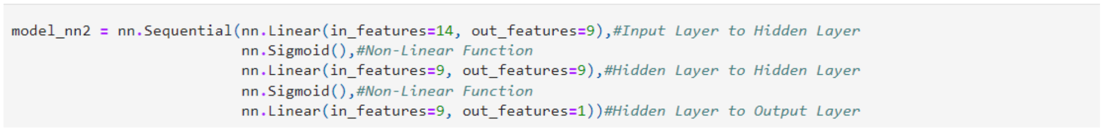


```
X_train_nn2 = torch.from_numpy(X_train_nn2).type(torch.float32)
Y_train_nn2 = torch.from_numpy(Y_train_nn2.to_numpy()).type(torch.float32)
X_test_nn2 = torch.from_numpy(X_test_nn2).type(torch.float32)
Y_test_nn2 = torch.from_numpy(Y_test_nn2.to_numpy()).type(torch.float32)
X_train_nn2[:5],Y_train_nn2[:5]
```

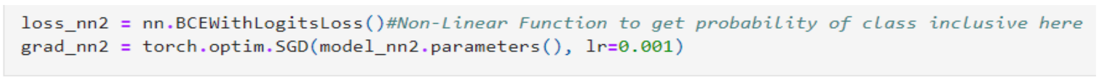


```
X_test_nn2, X_valid_nn2, Y_test_nn2, Y_valid_nn2 = train_test_split(X_test_nn2, Y_test_nn2, train_size=0.5, random_state=0)
X_test_nn2.shape, X_valid_nn2.shape
```

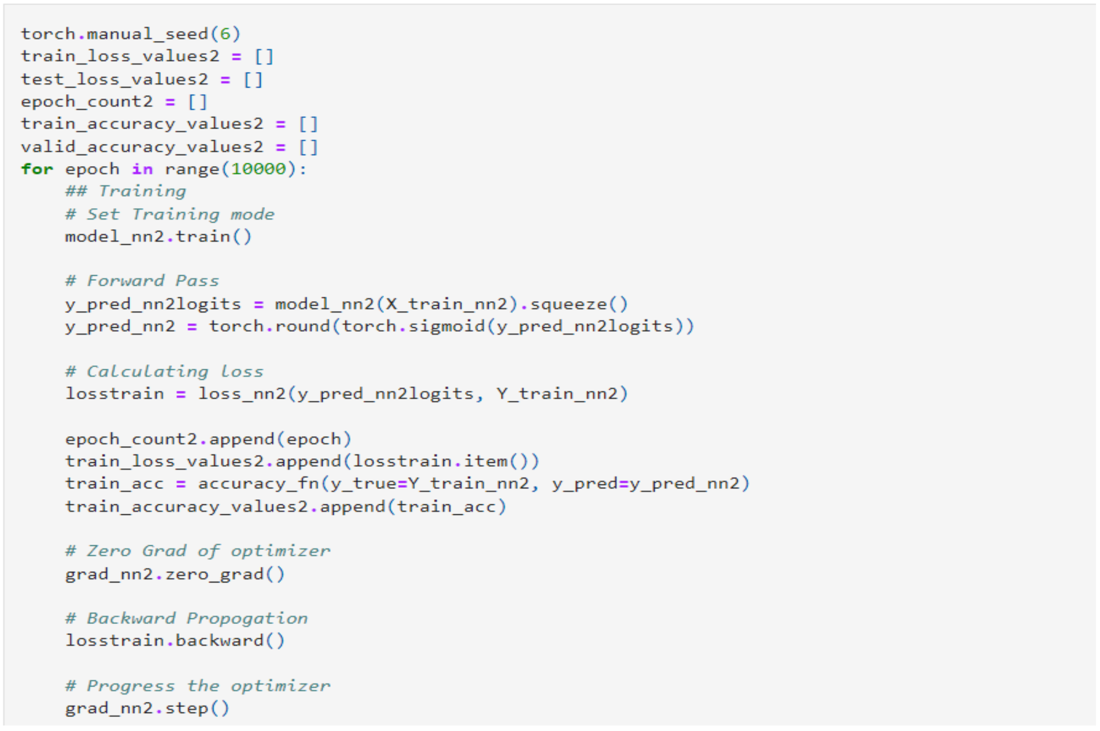
```
(torch.Size([8140, 14]), torch.Size([8141, 14]))
```



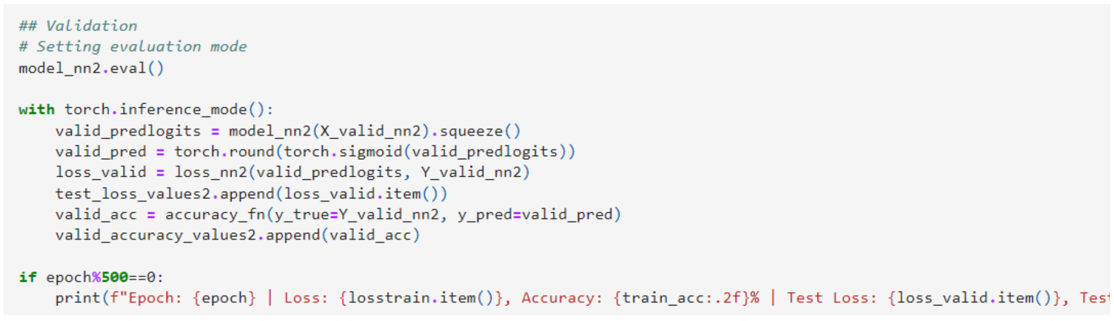
```
model_nn2 = nn.Sequential(nn.Linear(in_features=14, out_features=9),#Input Layer to Hidden Layer
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer to Hidden Layer
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=1))#Hidden Layer to Output Layer
```



```
loss_nn2 = nn.BCEWithLogitsLoss()#Non-Linear Function to get probability of class inclusive here
grad_nn2 = torch.optim.SGD(model_nn2.parameters(), lr=0.001)
```



```
torch.manual_seed(6)
train_loss_values2 = []
test_loss_values2 = []
epoch_count2 = []
train_accuracy_values2 = []
valid_accuracy_values2 = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn2.train()

    # Forward Pass
    y_pred_nn2logits = model_nn2(X_train_nn2).squeeze()
    y_pred_nn2 = torch.round(torch.sigmoid(y_pred_nn2logits))

    # Calculating loss
    losstrain = loss_nn2(y_pred_nn2logits, Y_train_nn2)

    epoch_count2.append(epoch)
    train_loss_values2.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn2, y_pred=y_pred_nn2)
    train_accuracy_values2.append(train_acc)

    # Zero Grad of optimizer
    grad_nn2.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn2.step()
```

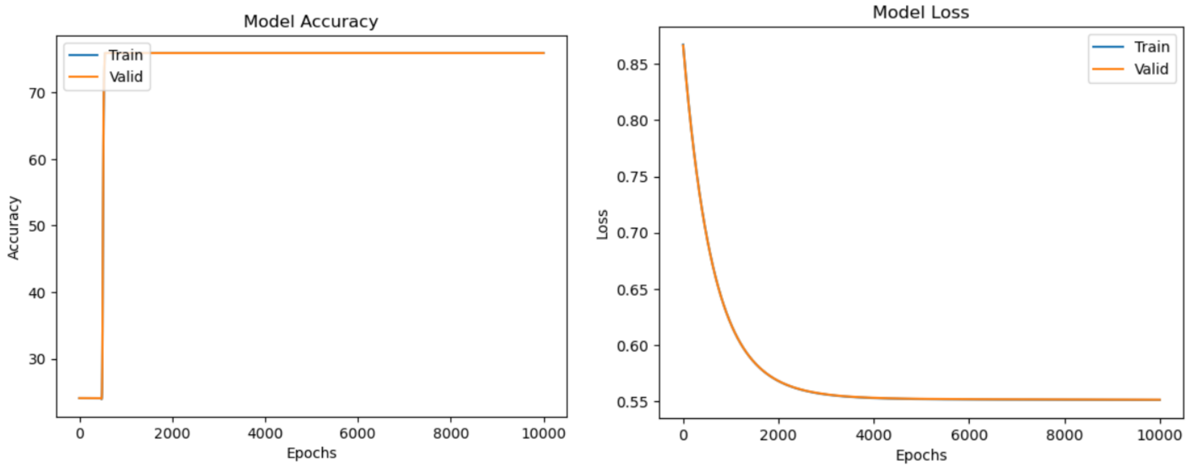


```
## Validation
# Setting evaluation mode
model_nn2.eval()

with torch.inference_mode():
    valid_predlogits = model_nn2(X_valid_nn2).squeeze()
    valid_pred = torch.round(torch.sigmoid(valid_predlogits))
    loss_valid = loss_nn2(valid_predlogits, Y_valid_nn2)
    test_loss_values2.append(loss_valid.item())
    valid_acc = accuracy_fn(y_true=Y_valid_nn2, y_pred=valid_pred)
    valid_accuracy_values2.append(valid_acc)

if epoch%500==0:
    print(f"Epoch: {epoch} | Loss: {losstrain.item()}, Accuracy: {train_acc:.2f}% | Test Loss: {loss_valid.item()}, Tes
model_nn2.eval()
with torch.inference_mode():
    train_pred = model_nn2(X_train_nn2).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn2, y_pred=train_pred)
train_acc
```

```
75.91904425539757
```

```
model_nn2.eval()
with torch.inference_mode():
    test_pred = model_nn2(X_test_nn2).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn2, y_pred=test_pred)
test_acc
```

```
76.85503685503686
```



The accuracy with 2 hidden layers is 75.9% on the training dataset and 76.8% on the testing dataset.

3.4.3 With 3 Hidden Layers

Similar steps as in the case with 1 Hidden layer are repeated. Here, the Input layer has 14 input features and 9 output features that become the input of Hidden Layer 1. This layer also has 9 output features that become the input of Hidden Layer 2. The 9 output features are input for the Hidden Layer 3. These features give 9 outputs that act as inputs for the Output layer with 1 final output feature.

```
X_train_nn3 = torch.from_numpy(X_train_nn3).type(torch.float32)
Y_train_nn3 = torch.from_numpy(Y_train_nn3.to_numpy()).type(torch.float32)
X_test_nn3 = torch.from_numpy(X_test_nn3).type(torch.float32)
Y_test_nn3 = torch.from_numpy(Y_test_nn3.to_numpy()).type(torch.float32)
X_train_nn3[:5],Y_train_nn3[:5]
```

```
model_nn3 = nn.Sequential(nn.Linear(in_features=14, out_features=9),#Input Layer to Hidden Layer1
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer1 to Hidden Layer2
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer2 to Hidden Layer3
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=1))#Hidden Layer3 to Output Layer
```

```
loss_nn3 = nn.BCEWithLogitsLoss()#Non-Linear Function to get probability of class inclusive here
grad_nn3 = torch.optim.SGD(model_nn3.parameters(), lr=0.001)
```

```
torch.manual_seed(6)
train_loss_values3 = []
test_loss_values3 = []
epoch_count3 = []
train_accuracy_values3 = []
valid_accuracy_values3 = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn3.train()

    # Forward Pass
    y_pred_nn3logits = model_nn3(X_train_nn3).squeeze()
    y_pred_nn3 = torch.round(torch.sigmoid(y_pred_nn3logits))

    # Calculating loss
    losstrain = loss_nn3(y_pred_nn3logits, Y_train_nn3)

    epoch_count3.append(epoch)
    train_loss_values3.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn3, y_pred=y_pred_nn3)
    train_accuracy_values3.append(train_acc)

    # Zero Grad of optimizer
    grad_nn3.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn3.step()
```

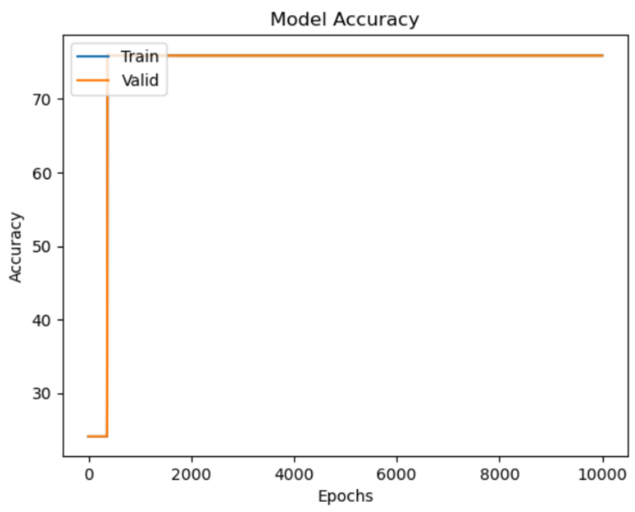
```
## Validation
# Setting evaluation mode
model_nn3.eval()

with torch.inference_mode():
    valid_predlogits = model_nn3(X_valid_nn3).squeeze()
    valid_pred = torch.round(torch.sigmoid(valid_predlogits))
    loss_valid = loss_nn3(valid_predlogits, Y_valid_nn3)
    test_loss_values3.append(loss_valid.item())
    valid_acc = accuracy_fn(y_true=Y_valid_nn3, y_pred=valid_pred)
    valid_accuracy_values3.append(valid_acc)

if epoch%500==0:
    print(f"Epoch: {epoch} | Loss: {losstrain.item()}, Accuracy: {train_acc:.2f}% | Test Loss: {loss_valid.item()}, Test
```

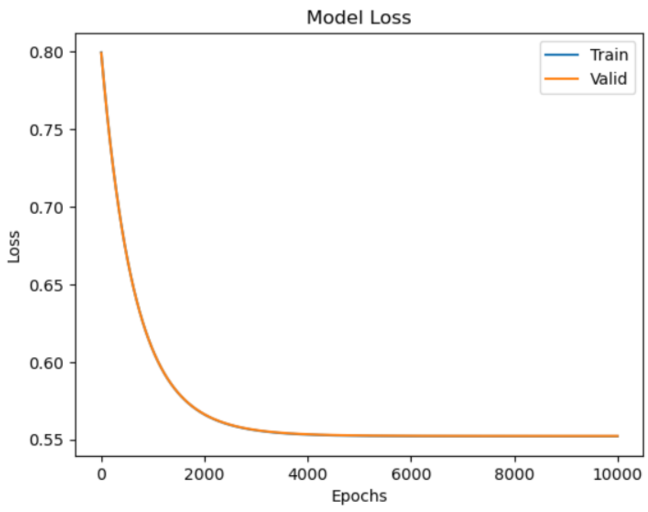
```
model_nn3.eval()
with torch.inference_mode():
    train_pred = model_nn3(X_train_nn3).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn3, y_pred=train_pred)
train_acc
```

```
75.91904425539757
```

```
model_nn3.eval()
with torch.inference_mode():
    test_pred = model_nn3(X_test_nn3).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn3, y_pred=test_pred)
test_acc
```

```
76.85503685503686
```





.

The accuracy with 3 hidden layers is 75.9% on the training dataset and 76.8% on the testing dataset.

4. Random Data

Here, we will use the 'train-test split' function. The data is first combined and then randomly selected 67% for training and 33% for testing. This is to check whether the data is well distributed or not. The pre-processing of the datasets is the same as shown previously.

4.1 Random Naive Bayes Classification

```
Y_pred_nbr_ = naiveGaussRandom.predict(X_train_nbr)
cm_nbr_ = confusion_matrix(Y_train_nbr, Y_pred_nbr_)
```

```
Y_pred_nbr = naiveGaussRandom.predict(X_test_nbr)
cm_nbr = confusion_matrix(Y_test_nbr, Y_pred_nbr)
```

```
naiveGaussRandom.score(X_train_nbr, Y_train_nbr)
```

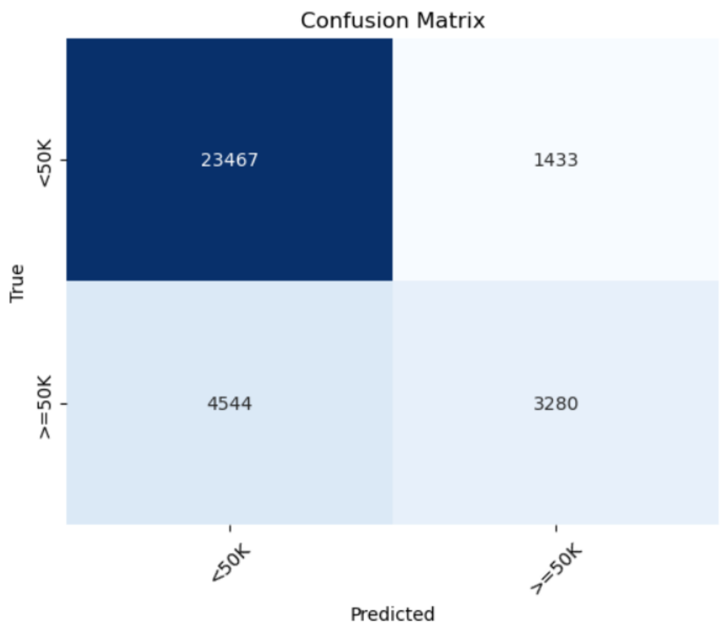
```
0.8173511795624007
```

```
naiveGaussRandom.score(X_test_nbr, Y_test_nbr)
```

```
0.8164784712743517
```

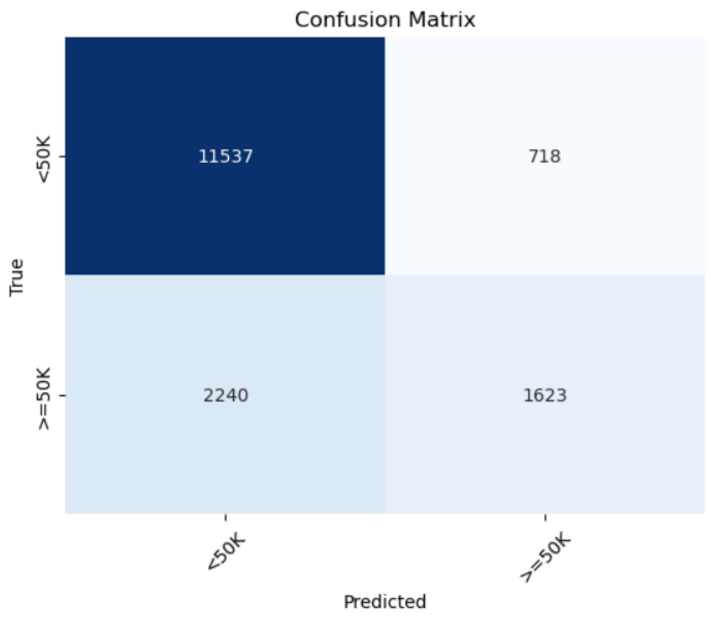
```
class_labels = ['<50K', '>=50K']
fig_nbr, ax_nbr = plt.subplots()

# Create a heatmap using seaborn
heatmap = sns.heatmap(cm_nbr_, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax_nbr)

ax_nbr.set_xlabel('Predicted')
ax_nbr.set_ylabel('True')
ax_nbr.set_title('Confusion Matrix')
ax_nbr.xaxis.set_ticklabels(class_labels)
ax_nbr.yaxis.set_ticklabels(class_labels)
plt.xticks(rotation=45)
```

```
(array([0.5, 1.5]), [Text(0.5, 0, '<50K'), Text(1.5, 0, '>=50K')])
```



```
class_labels = ['<50K', '>=50K']
fig_nbr1, ax_nbr1 = plt.subplots()

# Create a heatmap using seaborn
heatmap = sns.heatmap(cm_nbr, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax_nbr1)

ax_nbr1.set_xlabel('Predicted')
ax_nbr1.set_ylabel('True')
ax_nbr1.set_title('Confusion Matrix')
ax_nbr1.xaxis.set_ticklabels(class_labels)
ax_nbr1.yaxis.set_ticklabels(class_labels)
fig_nbr1.savefig('CM_NaiveRandom.png')
plt.xticks(rotation=45)
```

```
(array([0.5, 1.5]), [Text(0.5, 0, '<50K'), Text(1.5, 0, '>=50K')])
```

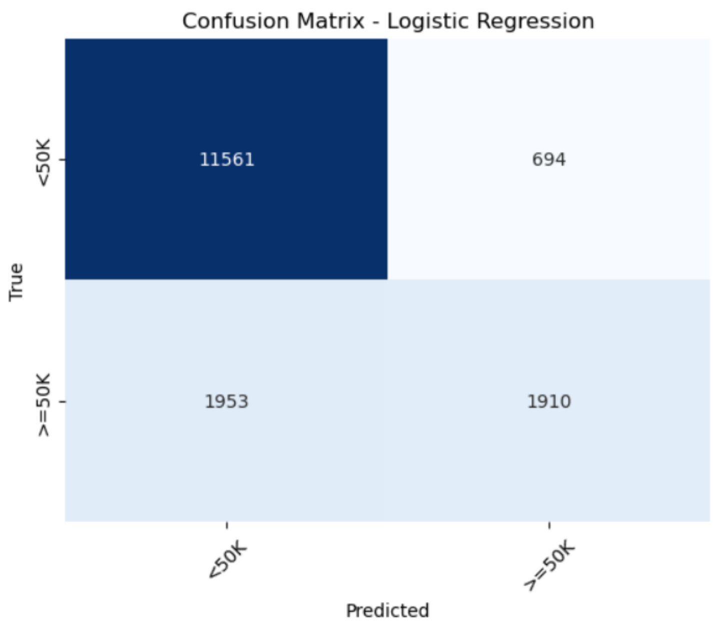


As seen from the above output, the accuracy is 81.7% on training data and 81.64% on testing data.

4.2 Logistic Regression with Random Data

```
logRegRandom = LogisticRegression(penalty=None, random_state=0,max_iter=500, multi_class='ovr',n_jobs=4)
X_train_lrR, Y_train_lrR, X_test_lrR, Y_test_lrR = X_train.copy(), Y_train.copy(), X_test.copy(), Y_test.copy()
logRegRandom.fit(X_train_lrR, Y_train_lrR)
```

```
Y_pred_lrR = logRegRandom.predict(X_test_lrR)
cm_lrR = confusion_matrix(Y_test_lrR, Y_pred_lrR)
class_labels = ['<50K', '>=50K']
fig_lrR, ax_lrR = plt.subplots()

# Create a heatmap using seaborn
heatmap = sns.heatmap(cm_lrR, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax_lrR)

ax_lrR.set_xlabel('Predicted')
ax_lrR.set_ylabel('True')
ax_lrR.set_title('Confusion Matrix - Logistic Regression')
```



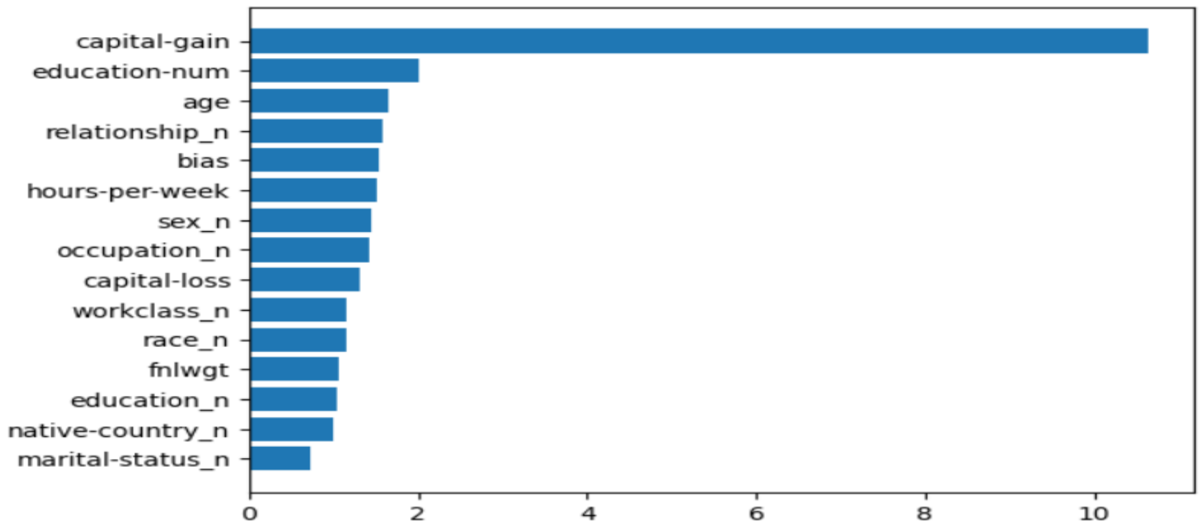
```
logRegRandom.score(X_train_lrR, Y_train_lrR)
```

```
0.8316831683168316
```

```
logRegRandom.score(X_test_lrR, Y_test_lrR)
```

```
0.8357736691897257
```

```
cw = logRegRandom.coef_[0]
cw0 = logRegRandom.intercept_[0]
feature_importance_lrR = pd.DataFrame(train_df.drop(columns='class_n', axis='columns').columns, columns = ["attribute"])
feature_importance_lrR["importance"] = pow(math.e, cw)

bias_row = pd.DataFrame({'attribute':['bias'],
                         'importance':[abs(cw0)]}, index=[0])
feature_importance_lrR = pd.concat([feature_importance_lrR, bias_row], ignore_index=True)
feature_importance_lrR = feature_importance_lrR.sort_values(by = ["importance"], ascending=False)

fig_lrfiR, ax_lrfiR = plt.subplots()
ax_lrfiR.barh(feature_importance_lrR.attribute, feature_importance_lrR.importance)
ax_lrfiR.invert_yaxis()
```



$$y = 0.50239895x_{14} + 0.06241798x_{13} + 0.69972453x_{12} + 2.36553087x_{11} + 0.27625578x_{10} + 0.40679473x_9 + 0.14577416x_8$$

$$+ 0.03022086x_7 - 0.33564394x_6 + 0.34494034x_5 + 0.45734211x_4 + 0.13318496x_3 + 0.36153185x_2 + 0.36153185x - 1.534844087484$$

where x represents as follows: 14:age, 13:fnlwgt, 12:education-num, 11:capital-gain, 10:capital-loss, 9:hours-per-week, 8:workclass_n, 7:education_n, 6:marital-status_n, 5:occupation_n, 4:relationship_n, 3:race_n, 2:sex_n, 1:native-country_n, 0:bias

4.3 Random Data Neural Network with 1 Hidden Layer

The number of features in each layer are the same as that of the above neural network model with one hidden layer.

```
torch.manual_seed(6)
train_loss_values = []
test_loss_values = []
epoch_count = []
train_accuracy_values = []
valid_accuracy_values = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn1R.train()

    # Forward Pass
    y_pred_nn1logits = model_nn1R(X_train_nn1).squeeze()
    y_pred_nn1 = torch.round(torch.sigmoid(y_pred_nn1logits))

    # Calculating loss
    losstrain = loss_nn1R(y_pred_nn1logits, Y_train_nn1)

    epoch_count.append(epoch)
    train_loss_values.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn1, y_pred=y_pred_nn1)
    train_accuracy_values.append(train_acc)

    # Zero Grad of optimizer
    grad_nn1R.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn1R.step()

    ## Validation
    # Setting evaluation mode
    model_nn1R.eval()

    with torch.inference_mode():
        valid_predlogits = model_nn1R(X_valid_nn1).squeeze()
        valid_pred = torch.round(torch.sigmoid(valid_predlogits))
        loss_valid = loss_nn1R(valid_predlogits, Y_valid_nn1)
        test_loss_values.append(loss_valid.item())
        valid_acc = accuracy_fn(y_true=Y_valid_nn1, y_pred=valid_pred)
        valid_accuracy_values.append(valid_acc)

    if epoch%500==0:
        print(f"Epoch: {epoch} | Loss: {losstrain.item()}, Accuracy: {train_acc:.2f}% | Test Loss: {loss_valid.item()}, Tes
```

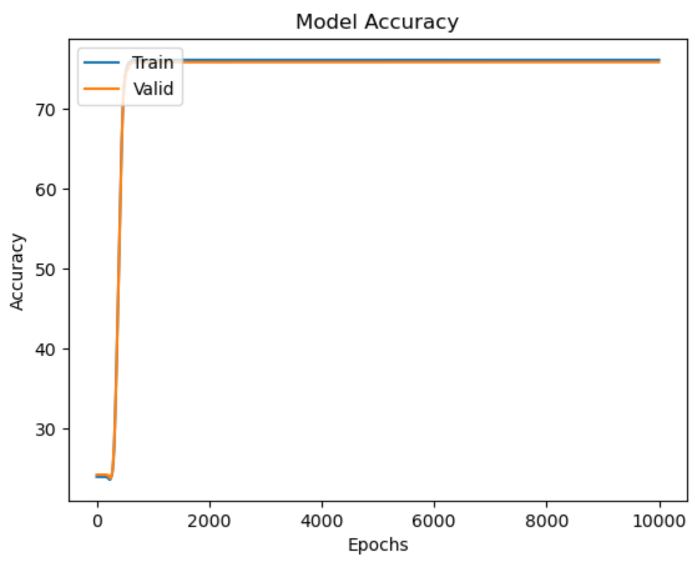
The data is trained and validated similarly using loss functions and gradient descent optimizers. The loss and accuracy are stored in respective arrays.

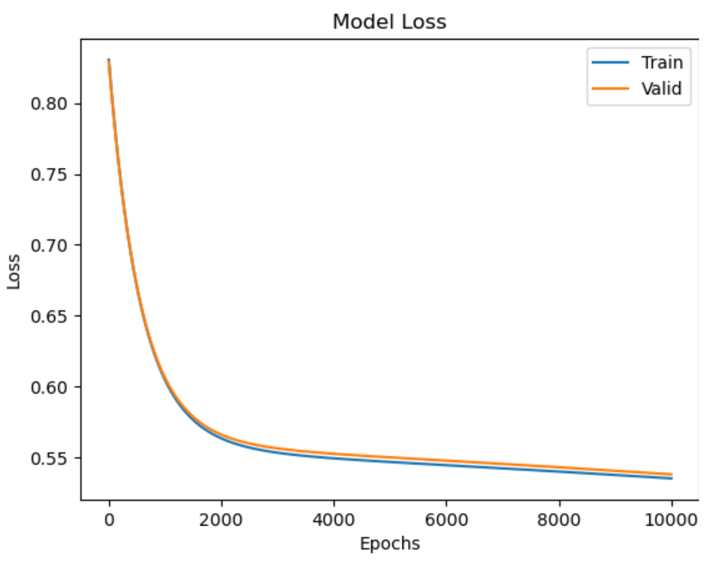
```
model_nn1R.eval()
with torch.inference_mode():
    train_pred = model_nn1R(X_train_nn1).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn1, y_pred=train_pred)
cm_nn1R_train = confusion_matrix(Y_train_nn1, train_pred)
train_acc
```

```
76.0909424275761
```

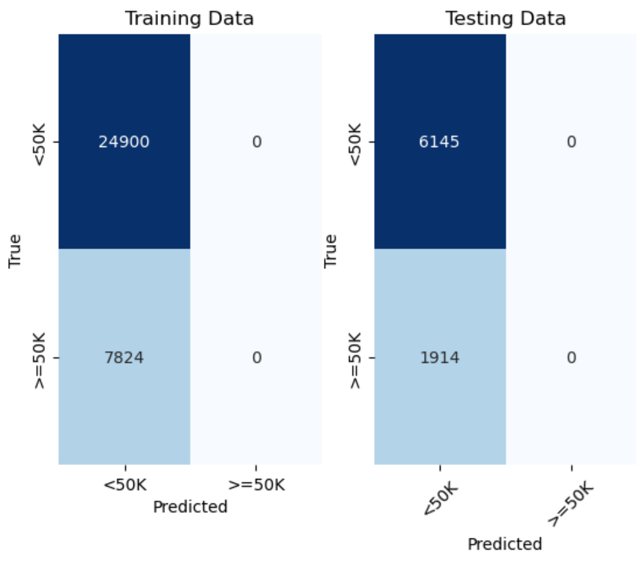
```
model_nn1R.eval()
with torch.inference_mode():
    test_pred = model_nn1R(X_test_nn1).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn1, y_pred=test_pred)
cm_nn1R_test = confusion_matrix(Y_test_nn1, test_pred)
test_acc
```

```
76.25015510609256
```





```
fignn1R, axnn1R = plt.subplots(1,2)
heatmap = sns.heatmap(cm_nn1R_train, annot=True, fmt='d', cmap='Blues', cbar=False, ax=axnn1R[0])

axnn1R[0].set_xlabel('Predicted')
axnn1R[0].set_ylabel('True')
axnn1R[0].xaxis.set_ticklabels(class_labels)
axnn1R[0].yaxis.set_ticklabels(class_labels)
plt.xticks(rotation=45)

heatmap1 = sns.heatmap(cm_nn1R_test, annot=True, fmt='d', cmap='Blues', cbar=False, ax=axnn1R[1])

axnn1R[1].set_xlabel('Predicted')
axnn1R[1].set_ylabel('True')
axnn1R[1].xaxis.set_ticklabels(class_labels)
axnn1R[1].yaxis.set_ticklabels(class_labels)
```



4.4 Random Data Neural Network with 2 Hidden Layers

```
model_nn2R = nn.Sequential(nn.Linear(in_features=14, out_features=9),#Input Layer to Hidden Layer
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer to Hidden Layer
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=1))#Hidden Layer to Output Layer
```

```
loss_nn2R = nn.BCEWithLogitsLoss()#Non-Linear Function to get probability of class inclusive here
grad_nn2R = torch.optim.SGD(model_nn2R.parameters(), lr=0.001)
```

```
torch.manual_seed(6)
train_loss_values2 = []
test_loss_values2 = []
epoch_count2 = []
train_accuracy_values2 = []
valid_accuracy_values2 = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn2R.train()

    # Forward Pass
    y_pred_nn2logits = model_nn2R(X_train_nn2R).squeeze()
    y_pred_nn2 = torch.round(torch.sigmoid(y_pred_nn2logits))

    # Calculating loss
    losstrain = loss_nn2R(y_pred_nn2logits, Y_train_nn2R)

    epoch_count2.append(epoch)
    train_loss_values2.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn2R, y_pred=y_pred_nn2)
    train_accuracy_values2.append(train_acc)

    # Zero Grad of optimizer
    grad_nn2R.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn2R.step()
```

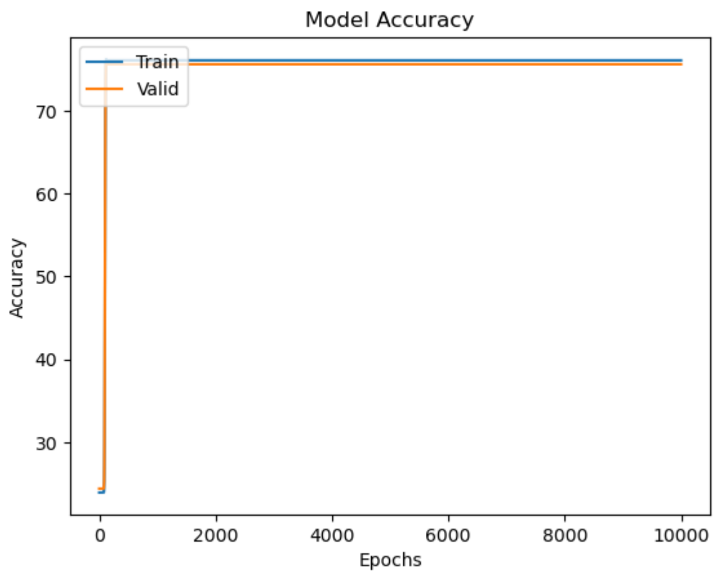
```
## Validation
# Setting evaluation mode
model_nn2R.eval()

with torch.inference_mode():
    valid_predlogits = model_nn2R(X_valid_nn2R).squeeze()
    valid_pred = torch.round(torch.sigmoid(valid_predlogits))
    loss_valid = loss_nn2R(valid_predlogits, Y_valid_nn2R)
    test_loss_values2.append(loss_valid.item())
    valid_acc = accuracy_fn(y_true=Y_valid_nn2R, y_pred=valid_pred)
    valid_accuracy_values2.append(valid_acc)
```

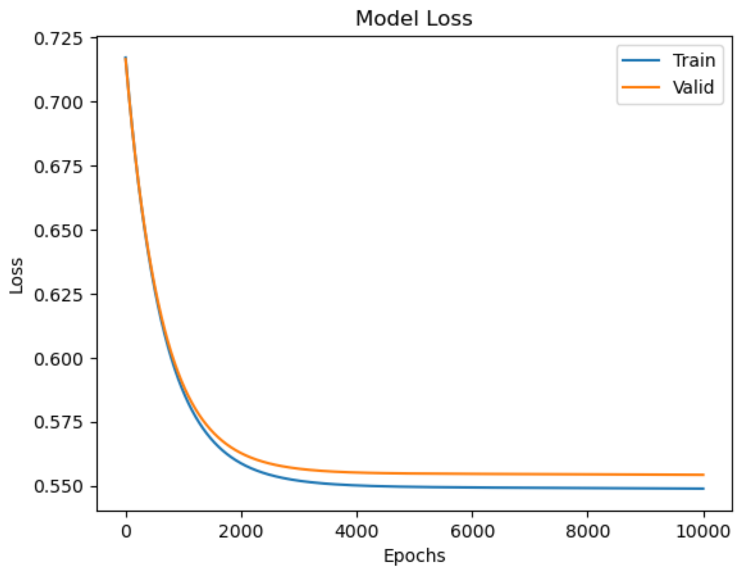
```
model_nn2R.eval()
with torch.inference_mode():
    train_pred = model_nn2R(X_train_nn2R).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn2R, y_pred=train_pred)
cm_nn2R_train = confusion_matrix(Y_train_nn2R, train_pred)
train_acc
```

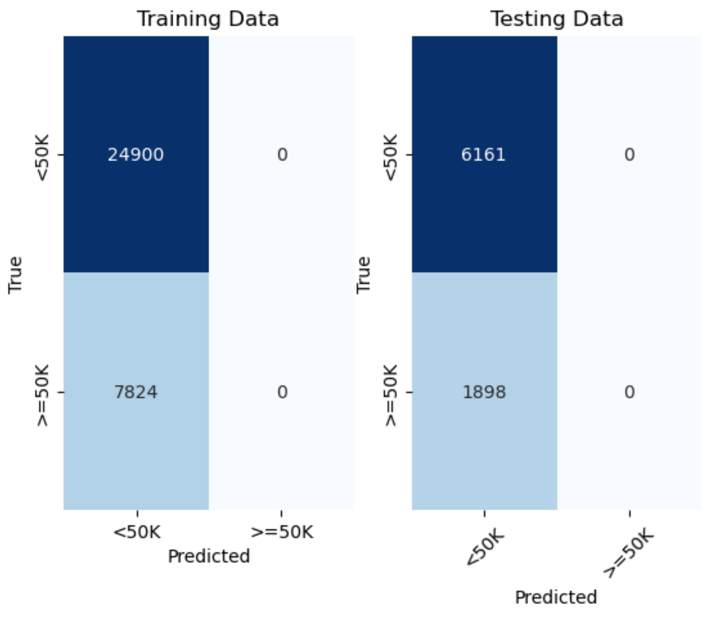
```
76.0909424275761
```

```
model_nn2R.eval()
with torch.inference_mode():
    test_pred = model_nn2R(X_test_nn2R).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn2R, y_pred=test_pred)
cm_nn2R_test = confusion_matrix(Y_test_nn2R, test_pred)
test_acc
```

```
76.44869090457873
```




Model Loss
Train
Valid
Loss
0.725
0.700
0.675
0.650
0.625
0.600
0.575
0.550
0
2000
4000
6000
8000
10000
Epochs


Training Data
Testing Data
True
<50K
>=50K
24900
0
7824
0
6161
0
1898
0
Predicted

4.5 Random Data Neural Network with 3 Hidden Layers

```
model_nn3R = nn.Sequential(nn.Linear(in_features=14, out_features=9),#Input Layer to Hidden Layer1
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer1 to Hidden Layer2
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=9),#Hidden Layer2 to Hidden Layer3
                          nn.Sigmoid(),#Non-Linear Function
                          nn.Linear(in_features=9, out_features=1))#Hidden Layer3 to Output Layer
```

```
loss_nn3R = nn.BCEWithLogitsLoss()#Non-Linear Function to get probability of class inclusive here
grad_nn3R = torch.optim.SGD(model_nn3R.parameters(), lr=0.001)
```

```
torch.manual_seed(6)
train_loss_values3 = []
test_loss_values3 = []
epoch_count3 = []
train_accuracy_values3 = []
valid_accuracy_values3 = []
for epoch in range(10000):
    ## Training
    # Set Training mode
    model_nn3R.train()

    # Forward Pass
    y_pred_nn3logits = model_nn3R(X_train_nn3R).squeeze()
    y_pred_nn3 = torch.round(torch.sigmoid(y_pred_nn3logits))

    # Calculating Loss
    losstrain = loss_nn3R(y_pred_nn3logits, Y_train_nn3R)

    epoch_count3.append(epoch)
    train_loss_values3.append(losstrain.item())
    train_acc = accuracy_fn(y_true=Y_train_nn3R, y_pred=y_pred_nn3)
    train_accuracy_values3.append(train_acc)

    # Zero Grad of optimizer
    grad_nn3R.zero_grad()

    # Backward Propogation
    losstrain.backward()

    # Progress the optimizer
    grad_nn3R.step()
```

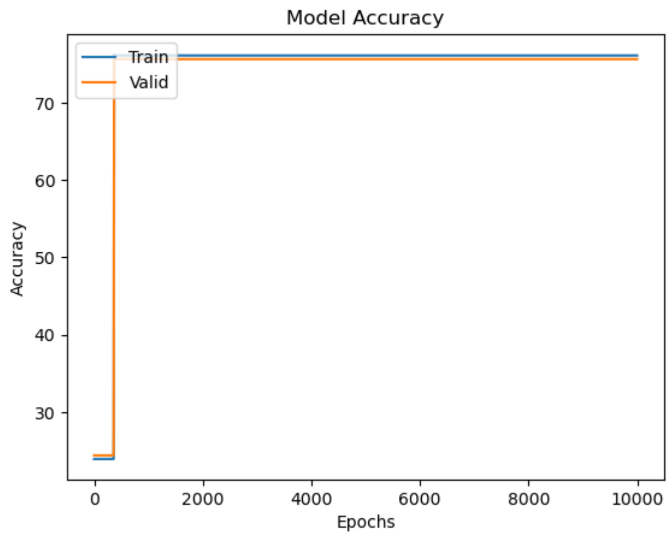
```
## Validation
# Setting evaluation mode
model_nn3R.eval()

with torch.inference_mode():
    valid_predlogits = model_nn3R(X_valid_nn3R).squeeze()
    valid_pred = torch.round(torch.sigmoid(valid_predlogits))
    loss_valid = loss_nn3R(valid_predlogits, Y_valid_nn3R)
    test_loss_values3.append(loss_valid.item())
    valid_acc = accuracy_fn(y_true=Y_valid_nn3R, y_pred=valid_pred)
    valid_accuracy_values3.append(valid_acc)
```

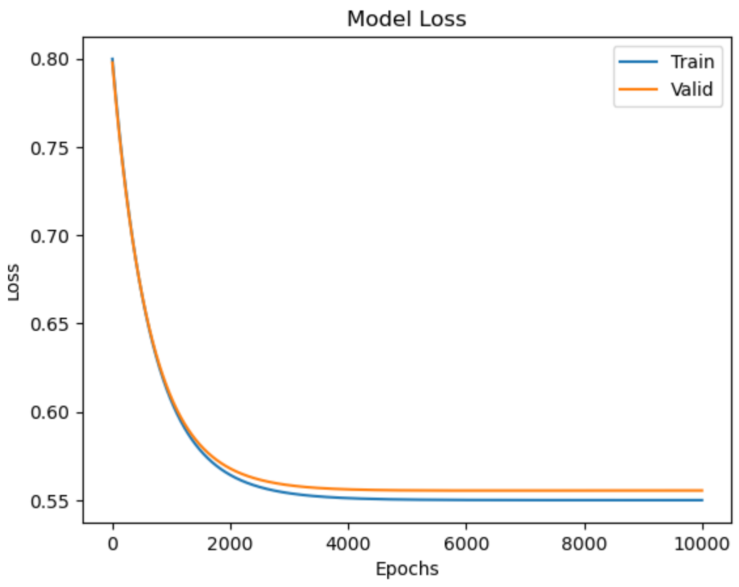
```
model_nn3R.eval()
with torch.inference_mode():
    train_pred = model_nn3R(X_train_nn3R).squeeze()

train_pred = torch.round(torch.sigmoid(train_pred))
train_acc = accuracy_fn(y_true=Y_train_nn3R, y_pred=train_pred)
cm_nn3R_train = confusion_matrix(Y_train_nn3R, train_pred)
train_acc
```

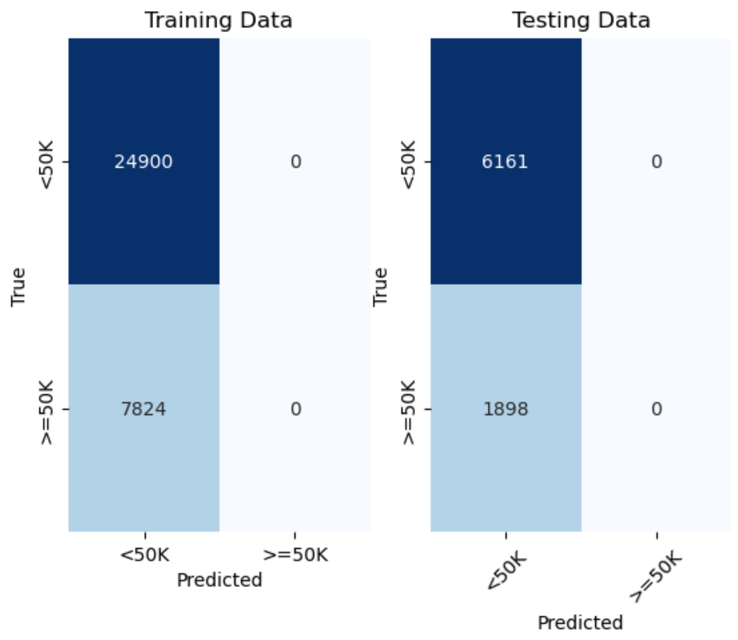
```
76.0909424275761
```

```
model_nn3R.eval()
with torch.inference_mode():
    test_pred = model_nn3R(X_test_nn3R).squeeze()

test_pred = torch.round(torch.sigmoid(test_pred))
test_acc = accuracy_fn(y_true=Y_test_nn3R, y_pred=test_pred)
cm_nn3R_test = confusion_matrix(Y_test_nn3R, test_pred)
test_acc
```

```
76.44869090457873
```







5. Conclusion

The results for all the models we have created so far give us some insight into the data and its distribution. The training and testing data have more <50K class and the lack of >=50K class creates a problem for the model to recognize it ideally. This can be demonstrated using the confusion matrix for training and testing data. Comparing the Confusion Matrices for training and testing data, we can clearly observe that the matrix correctly classifies the <50K class. In contrast, it fails to do so accurately for the >=50K class. The reason is also apparent: the overall data has a low amount of training for >=50K class, causing the model to fail in learning the generality of the data.

Naive Bayes Classifier

Obtaining a score of 0.8192008844937195 for training and 0.8161046618758061 for testing data, this model can correctly predict more of >=50K classes compared to <50K classes. But if we randomly take data for training and testing in the combined data, we observe 0.8173511795624007 and 0.8164784712743517 for training and testing data, which is only a minuscule improvement.

Logistic Regression Model

Moving to Logistic Regression Model, we obtain 0.8327139829857806 and 0.8300472943922363 for training and testing on the given dataset, which is better than the NB model. It also shows similar results but is better in classifying> =50K class. For Random datasets, we get 0.8316831683168316 and 0.8357736691897257 for training and testing, which is a 0.5% improvement in testing, showing that the random dataset is better in generalizing the result.

Neural Network

For the given dataset, we get 75.91904425539757 and 76.8058968058968 for training and testing. We observe that the confusion matrix basically predicts the >=50K class to be 0. This is not an accurate description, as the Neural Network returns the probability of data, and we have forcibly rounded them off to comprehend the data, resulting in the low probabilities being converted to 0. We're left with this output since we don't have another metric with our current knowledge to quantify our result. So, for the random dataset, we get 76.0909424275761 and 76.25015510609256 for training and testing, which is constant through the three-layer changes, so how do we quantify the usefulness of hidden layers?

We will look at the data yielded while training: the loss and accuracy graphs. These graphs become deeper and stabler as the no. of hidden layers increases. We can also observe that as we go from given to random data, the graph for validation and training data indicates, allowing us to decide where to stop training to get a more generalized result.

Another method is to change our threshold for probability calculation from 0.5 to a ratio that represents testing/training data, but the validity of this method needs to be clarified.

6. Result: Ranking the classifiers

So far, we have classified the given data to get final salary classes using five methods. Among those, the efficacy of the Decision Tree has been the highest, with an accuracy of 85.92% on testing data, given we perform post-pruning. Following that, with an accuracy of 85.47% on testing data, is Random Forest Classifier. Following is the logistic regression model, with an accuracy of 83.58%, followed by the Naive Bayes Classifier, with an accuracy of 81.65%. Neural Network, in general, ranked the lowest with accuracy, only reaching 76.81%, but this is arguable since our method of accuracy calculation is not verified. To finalize, we can say that our models have been Ranked as:

- Decision Tree
- Random Forest
- Logistic Regression
- Naive Bayes
- Neural Network,

but if we can obtain correct data, which has an equal amount of both classes or correct accuracy threshold, we can with guarantee state that Neural Network can give the most general result, which can be proven from the fact that while there is a big gap in the training and testing accuracy for all the other models, the training accuracy and testing accuracy of Neural Networks are similar, showing the generality of the result.

```
model_nn3R.eval()
with torch.inference_mode():
    test_pred = model_nn3R(X_test_nn3R).squeeze()
results = {'0.4':0,'0.3':0,'0.2':0,'0.1':0,'0.0':0}
test_pred = (torch.sigmoid(test_pred))
for i in range(0,len(Y_test_nn3R)):
    if Y_test_nn3R[i]==1:
        if test_pred[i]>=0.4:
            results['0.4']+=1
        elif test_pred[i]>=0.3:
            results['0.3']+=1
        elif test_pred[i]>=0.2:
            results['0.2']+=1
        elif test_pred[i]>=0.1:
            results['0.1']+=1
        else:
            results['0.0']+=1
results
```

```
{'0.4': 0, '0.3': 0, '0.2': 1898, '0.1': 0, '0.0': 0}
```

The above cell clearly shows that all answers lie between 0.2 and 0.3, which cannot be dismissed as a coincidence, and that the study needs to be further improved to improve our data. So, for now, we can simply say that a pruned decision tree is the best classifier for this uneven data distribution, whereas the Neural Network gives the most general distribution.