DM Assignment 2

Jai Anand Girkar

B01015181

Q1:

Steps taken for Data Cleaning:

Data was obtained from two files: **adult.data** and **adult.test**.

Initial exploration was conducted to understand the structure of the dataset.

Adding the column headers to the data set:

```
l1 = ['page','workclass','fnlwgt','education','education-num',
      'marital-status','occupation','relationship','race','sex','capital-gain',
      'capital-loss','hours-per-week','native-country', 'y']

df = df.rename(columns=dict(zip(df.columns, l1)))
```

Dimensionality Assessment: The dimensionality of the dataset was assessed using the **df.shape** function to determine the number of rows and columns.

Missing Values Analysis: - The total number of missing values (NaN) in the dataset was calculated to identify any potential data gaps.

```
print(f"adult.data shape : {adultdata.shape}")
print(f"adult.test shape : {adulttest.shape}")
```

```
adult.data shape : (32561, 15)
adult.test shape : (16281, 15)
```

```
na_data = adultdata.isna().sum().sum()
na_test = adulttest.isna().sum().sum()
```

Unique Values Examination: - Each column of the dataset was examined to identify unique values present and was compared with the feature values given in the **adult.names** file to remove any unknown or unidentified values.

```
distinct_values_per_column = {}
for column in obj_df.columns:
    distinct_values_per_column[column] = obj_df[column].unique()

# Display the distinct values from each column
for column, values in distinct_values_per_column.items():
    print(f"Distinct values in column '{column}':")
    print(values)
```

Handling Unknown Values: **?** values in each column were identified to denote unknown data. A filtered dataset was generated to replace **?** values with **mode** value of that particular column, ensuring consistency with the feature values specified in the **adult.names** file.

```
columns_without_question_mark = df.columns[df.isin([' ?']).any()]

# Print the columns not containing ' ?' values
print("Columns containing ' ?' values:")
for column in columns_without_question_mark:
    print(column)
```

```
Columns containing ' ?' values:
workclass
occupation
native-country
```

```
columns_with_question_mark = adultdata.columns[adultdata.isin([' ?']).any()]

# Calculate mode for each numeric column
string_present = adultdata.isin([" ?"]).any()

for column in columns_with_question_mark:
    if adultdata[column].dtype == 'object':
        column_mode = adultdata[column].mode().iloc[0]
#         adultdata[column] = adultdata[column].replace(' ?', column_mode)
        adultdata.loc[adultdata[column] == ' ?', column] = column_mode

columns_with_question_mark = adulttest.columns[adulttest.isin([' ?']).any()]
for column in columns_with_question_mark:
    if adultdata[column].dtype == 'object':
        column_mode = adulttest[column].mode().iloc[0]
        adulttest.loc[adulttest[column] == ' ?', column] = column_mode
```

Cleaning y_test Data: Modified the y_test (adult.test) output field to remove the "." And replace it by by an empty string

```
# adulttest_filtered
# df_y = df.iloc[:, -1]  # Select all rows and all columns except the last one
# df_x = df.iloc[:, :-1]
x_train = adultdata_filtered.iloc[:, :-1]  # Select the first 32,561 rows for training
x_test = adulttest_filtered.iloc[:, :-1]

y_train = adultdata_filtered.iloc[:, -1]  # Select the first 32,561 rows for training
y_test  = adulttest_filtered.iloc[:, -1]

y_test = y_test.str.replace('.', '')
```

Encoding: Label Encoder was used to convert the categorical variables into numerical variables. This conversion is essential for machine learning algorithm that require input data to be in numerical format to perform effectively.

Normalization: Normalization was performed using **StandardScalar()** to scale numerical features to a similar range to prevent certain features from dominating others during analysis. It is futher used to prevent bias, improve model performance and improve convergence.

```
label_encoder = LabelEncoder()

x_train_encoded = x_train.copy()
x_test_encoded = x_test.copy()

y_train_encoded = y_train.copy()
y_test_encoded = y_test.copy()

for column in x_train.select_dtypes(include=['object']):
    x_train_encoded[column] = label_encoder.fit_transform(x_train[column])
    x_test_encoded[column] = label_encoder.fit_transform(x_test[column])


from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x_train_encoded = pd.DataFrame(scaler.fit_transform(x_train_encoded), columns=x_train_encoded.columns)
x_test_encoded = pd.DataFrame(scaler.fit_transform(x_test_encoded), columns=x_test_encoded.columns)

# x_train_encoded
clf = DecisionTreeClassifier()
clf.fit(x_train_encoded, y_train)

# Make predictions on the evaluation dataset
y_pred = clf.predict(x_test_encoded)
```

Q2.

The classification method used is **KNN** with the n_neighbor value set to 3

1. Create an instance of the KNN classifier with the desired hyperparameters. In this case, set **n_neighbors=3**
2. Fit the classifier to the training data using the **fit()** method.
3. Use the trained classifier to make predictions on the test data using the **predict()** method
4. Define a function (**cal_accuracy_error**) to calculate the accuracy and error rate, and then call this function with the actual and predicted labels, along with the classifier name (in this case, "KNN").

```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(x_train_encoded, y_train)

y_pred = clf.predict(x_test_encoded)
cal_accuracy_error(y_test, y_pred, "KNN")
```

```
Classification Error Rate KNN: 17.984153307536392
```

As per the definition of the **Classification Error Rate** the Error rate was calculated by the classification error rate is defined as the number of the truth incorrect labels reported by your method in the evaluation dataset divided by the number of the truth labels in the evaluation dataset.

The classification Error Rate with **n_neighbor=3** is **17.984**

```
from sklearn.metrics import accuracy_score
import numpy as np
def cal_accuracy_error(y_test, y_pred, str1):
    misclassified_instances = (y_test != y_pred).sum()
    total_instances = len(y_test)
    error_rate = misclassified_instances / total_instances
    print(f"Classification Error Rate {str1}: {error_rate *100}")
```

Q3.

Reference:

Algorithm	Error
C4.5	15.54
C4.5-auto	14.46
C4.5 rules	14.94
Voted ID3 (0.6)	15.64
Voted ID3 (0.8)	16.47
T2	16.84
1R	19.54
NBTree	14.1
CN2	16.0
HOODG	14.82
FSS Naive Bayes	14.05
IDTM (Decision table)	14.46
Naive-Bayes	16.12
Nearest-neighbor (1)	21.42
Nearest-neighbor (3)	20.35
OC1	15.04
Pebls	Crashed. Unknown why (bounds WERE increased)

Obtained KNN neighbor=3 Error Rate: **17.984**

Based on the comparison with the reported error rates for various well-known classification methods listed in the **adult.names**, classification error rate of **17.984** with **KNN** (n_neighbors=3) is higher than several algorithms such as C4.5, C4.5-auto, NBTree, FSS Naive Bayes, and IDTM (Decision table),T2 which have reported error rates ranging from **14.1% to 16.84%** and some are higher than the 17.984 mark**.** However, the error rate of **Nearest – Neighbor** with the **n_neighbor=3** value is given as **20.35 which is higher from my obtained error rate(17.984)**.

This is mainly because of the preprocessing techniques used on data like replacing unknown missing **values with mode, using encoding for categorical variables** and **normalization** mentioned in answer 1

However, the model can be further optimized by futher tuning the KNN model by increasing the **n_neighbor** values, **removing duplicate rows**, **hyper parameter tuning**

Most importantly, efficient **Scaling** in K-Nearest Neighbors is needed because it ensures that all features contribute equally to the distance computation. Since KNN relies on **measuring the distance** between data points to determine similarity, features with larger scales may dominate the distance calculation, leading to biased results.

Here the model is further optimized by testing the model on different n_neighbor value and found that **n_neighbor** = **20** is found to be giving comparitively less error rate.

```
for i in range(5, 25, 5):
    clf = KNeighborsClassifier(n_neighbors=i)
    clf.fit(x_train_encoded, y_train)

    y_pred = clf.predict(x_test_encoded)
    cal_accuracy_error(y_test, y_pred, "KNN")
```

```
Classification Error Rate KNN: 17.204102942079725
Classification Error Rate KNN: 16.350347030280695
Classification Error Rate KNN: 16.430194705484922
Classification Error Rate KNN: 16.086235489220563
```

The n_neigbor value for 20 futher decreases the error rate to **16.08**

```
clf = KNeighborsClassifier(n_neighbors=20)
clf.fit(x_train_encoded, y_train)

y_pred = clf.predict(x_test_encoded)
cal_accuracy_error(y_test, y_pred, "KNN")
```

```
Classification Error Rate KNN: 16.086235489220563
```

Removing Duplicate Data

```
adultdata_filtered = adultdata_filtered.drop_duplicates(keep='first')
adulttest_filtered = adulttest_filtered.drop_duplicates(keep='first')
```

Choosing an optimized neighbor KNN algorithm to enhance Performance:

brute-force = Straighfoward to compute the distance from neighboring points

KD-tree= Responsible to recusivel divide the space into smaller regions uses logN

Ball-tree=partitions the space into nested hyperspheres

Q4.

The dataset was downsampled to various percentages ranging from 50% to 90%, utilizing a random seed value of 22.

```
def downsampled_training(percent_val):
    adultdata_filtered_new = adultdata_filtered.sample(frac=percent_val, random_state=22)
    x_train = adultdata_filtered_new.iloc[:, :-1]
    y_train = adultdata_filtered_new.iloc[:, -1]

    print(f"training data down sampled for {percent_val*100}%, Shape: {adultdata_filtered_new.shape}")
    label_encoder = LabelEncoder()

    x_train_encoded = x_train.copy()
    x_test_encoded = x_test.copy()

    y_train_encoded = y_train.copy()
    y_test_encoded = y_test.copy()
```

```python
# print(f"Original training data shape:{adultdata_filtered.shape}")
from sklearn.model_selection import GridSearchCV
err_list = []
for i in [0.5, 0.6, 0.7, 0.8, 0.9]:
    x_train_encoded, x_test_encoded, y_train = downsampled_training(i)
    clf = KNeighborsClassifier(n_neighbors=3)
    clf.fit(x_train_encoded, y_train)
    y_pred = clf.predict(x_test_encoded)
    err_list.append(cal_accuracy_error(y_test, y_pred, "KNN"))
    print("-----------------------------------------------------")

print(f"Mean: {np.mean(err_list)}, Deviation : {round(np.std(err_list),4)}")
```

```
training data down sampled for 50.0%, Shape: (16268, 15)
Classification Error Rate KNN: 18.54308703396597
-----------------------------------------------------
training data down sampled for 60.0%, Shape: (19522, 15)
Classification Error Rate KNN: 18.174559302254163
-----------------------------------------------------
training data down sampled for 70.0%, Shape: (22776, 15)
Classification Error Rate KNN: 18.082427369326208
-----------------------------------------------------
training data down sampled for 80.0%, Shape: (26030, 15)
Classification Error Rate KNN: 18.0332903384313
-----------------------------------------------------
training data down sampled for 90.0%, Shape: (29283, 15)
Classification Error Rate KNN: 18.076285240464347
-----------------------------------------------------
Mean: 18.181929856888395, Deviation : 0.1864
```

Classification Error Rates:

50% → 18.54

60% → 18.17

70% → 18.08

80% → 18.03

90% → 18.07

Mean: **18.18** (This suggests that, on average, the classification error rate remains relatively consistent across varying proportions of the training data)

Deviation: **0.1864** (The standard deviation of about 0.1864 reflects how much the classification error rates vary from the average. A lower standard deviation suggests that these error rates are closely grouped around the average, indicating a consistent performance across the different downsampling scenarios.)

When the training data is reduced to 50%, the error rate tends to be higher. However, as the training sample size increases to 60%, 70%, 80%, and 90%, the error rate decreases gradually, although the but the change is relatively small and is more or less the same.

1. **This suggests that the dataset's features are sufficiently informative for making predictions, even with a smaller sample size**.

2. **The patterns in the data are relatively simple and easy for the model to learn, even with a smaller amount of training data after efficient preprocessing**
3. **The dataset may have low noise or minimal irrelevant information**
4. With a smaller training sample size, the model may have **higher bias**. This data relevantly has **an acceptable bias** hence it performs good with smaller sample size.
5. With more data for training, the model has more opportunities to learn and improve its performance, resulting in a lower error rate. However, there's a trade-off with computation time. **However, effective preprocessing, feature selection, and scaling can mitigate this and potentially lead to lower error rates even with smaller training samples.**
6. More-over testing can also be done on multiple random seed values to get optimal model performance

Q5.

One the same cleaned, encoded, standardized data, I am running a Deep Neural Network for classification by putting the following parameters.

An appropriate encoding is needed of the resulting attribute when accurate classification output is needed from a deep neural network

Hence the **y_train** and **y_test** were encoded on priority.

```
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.fit_transform(y_test)
```

Here the model was model is build and fitted with the appropriate parameters.

```
model = Sequential([
    Dense(256, activation='relu', input_shape=(x_test_encoded.shape[1],)),
    BatchNormalization(),  # Add batch normalization layer
    Dropout(0.4),  # Adjust dropout rate
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),
    Dense(1, activation='sigmoid')
])

# Compile the model with a lower learning rate and binary cross-entropy loss
optimizer = Adam(learning_rate=0.0001)  # Lower learning rate
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

# Train the model with 10 epochs and a larger batch size
history = model.fit(x_train_encoded, y_train_encoded, epochs=10, batch_size=128, validation_split=0.2)


y_pred = (model.predict(x_test_encoded) > 0.5).astype("int32")
accuracy = accuracy_score(y_test_encoded, y_pred)
error_rate = 1 - accuracy
print("-------------------------------------------------------------")
print("Classification Error Rate for Neural Network:", round(error_rate, 4))
```

1. I used an input layer containing **256 neurons** utilizing the **Relu** activation function, with the input shape determined by the number of features in the encoded test data.

2. After each hidden layer, I incorporated **batch normalization** layers to normalize activations.

3. Dropout layers with a dropout rate of **0.4** were added after each batch normalization layer

4. I included two hidden layers with **128** and **64** neurons respectively, both with the **Relu** activation function.

5. Lastly, the output layer is single neuron utilizing a sigmoid activation function used for binary classification

Step taken to compile and training the model:

1. I compiled the model using the Adam optimizer with a lower learning rate set to **0.0001**.

2. For loss calculation, I opted for binary cross-entropy, suitable for binary classification.

3. To evaluate model performance, I selected the accuracy metric.

4. The model underwent training using the encoded training data (**x_train_encoded** and **y_train_encoded**).

5. Traning was done for **10** epochs with a batch size **128**.

6. Performance validation was done by using the validation split of **0.2** of the encoded training set.

```
Epoch 1/10
184/184 [==============================] - 3s 8ms/step - loss: 0.8462 - accuracy: 0.5730 - val_loss: 0.6532 - val_accuracy: 0.6317
Epoch 2/10
184/184 [==============================] - 1s 6ms/step - loss: 0.6823 - accuracy: 0.6615 - val_loss: 0.5658 - val_accuracy: 0.7401
Epoch 3/10
184/184 [==============================] - 1s 6ms/step - loss: 0.6172 - accuracy: 0.6983 - val_loss: 0.5134 - val_accuracy: 0.7796
Epoch 4/10
184/184 [==============================] - 1s 7ms/step - loss: 0.5665 - accuracy: 0.7286 - val_loss: 0.4709 - val_accuracy: 0.8006
Epoch 5/10
184/184 [==============================] - 2s 9ms/step - loss: 0.5337 - accuracy: 0.7453 - val_loss: 0.4407 - val_accuracy: 0.8115
Epoch 6/10
184/184 [==============================] - 2s 9ms/step - loss: 0.5022 - accuracy: 0.7640 - val_loss: 0.4207 - val_accuracy: 0.8180
Epoch 7/10
184/184 [==============================] - 1s 6ms/step - loss: 0.4834 - accuracy: 0.7777 - val_loss: 0.4054 - val_accuracy: 0.8228
Epoch 8/10
184/184 [==============================] - 1s 6ms/step - loss: 0.4709 - accuracy: 0.7839 - val_loss: 0.3910 - val_accuracy: 0.8262
Epoch 9/10
184/184 [==============================] - 1s 6ms/step - loss: 0.4470 - accuracy: 0.7967 - val_loss: 0.3810 - val_accuracy: 0.8294
Epoch 10/10
184/184 [==============================] - 1s 6ms/step - loss: 0.4364 - accuracy: 0.8013 - val_loss: 0.3740 - val_accuracy: 0.8318
509/509 [==============================] - 1s 1ms/step
------------------------------------------------------------
Classification Error Rate for Neural Network: 0.17
```

Prediction:

1. These predictions were converted into binary values (0 or 1) by thresholding the output probabilities at **0.5.**
2. Finally, the error was calculated by taking the accuracy into consideration.
   Classification Error rate: **0.17**

Conclusion:

Deep neural networks outperform other classification methods due to their ability to handle complex relationships, learn hierarchical feature representations, model non-linear patterns, perform end-to-end learning, and utilize regularization techniques effectively.

The Deep neural networks gives a Classification Error Rate of **0.17** which substantially less than the Models and the error rates defined in **adult.names.** Hence, we can beat all the classic classifiers reported in **adult.names**.

Link:
https://colab.research.google.com/drive/1L0r1dgKZlAGou1TDKheJFw8su0NY_lwf#scrollTo=SAHaOLIsLCrK