Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 019
Code Title:	Emerging Technologies in CpE 2 - 2nd Semester

ACTIVITY NO.	**Assignment 8.1 : Saving Models**
Name	Dela Cruz, Irish
Section	CPE32S3
Date Performed:	04/10/2024
Date Submitted:	04/19/2024
Instructor:	Engr. Roman M. Richard

Introduction of Data

The "Census Income" dataset, a.k.a the "Adult" dataset, is a widely used dataset in the machine learning community, particularly for classification tasks. It contains demographic information from the US Census Bureau and is commonly used to predict whether an individual's income exceeds $50K per year based on various attributes

Link: https://archive.ics.uci.edu/dataset/2/adult

Dataset:

- Multivariate Subject Area:
- Social Science

Features: (categorical and integer type)

- age
- education level
- marital status
- occupation
- race
- gender
- capital gain
- capital loss
- houra worked per week
- native country

Instances:

- 48,842

This dataset is mostly used for task such as income prediction, demographic analysis, understanding the factors that influence income levels, and eploring data pre-processing techniques due to its real-world relevance and moderate size.

› Installing the requirement packages

[ ] ↳ *2 cells hidden*

⌄ Explonatory Data Analysis (EDA)

The following task were performed in this section:

1. Checking the data is properly loaded using head and tail
2. Checking the information of the dataframes
3. Checking the features by graphing all the values
4. Checking the values count of each variables

```
X.shape, y.shape
```

```
((48842, 14), (48842, 1))
```

```
X.head(10)
```

	age	workclass	fnlwgt	education	educati
0	39	State-gov	77516	Bachelors	
1	50	Self-emp-not-inc	83311	Bachelors	
2	38	Private	215646	HS-grad	
3	53	Private	234721	11th	
4	28	Private	338409	Bachelors	

Next steps: View recommended plots

```
y.head(10)
```

	income
0	<=50K
1	<=50K
2	<=50K
3	<=50K
4	<=50K
5	<=50K
6	<=50K
7	>50K
8	>50K
9	>50K

Next steps: View recommended plots

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   age             48842 non-null  int64 
 1   workclass       47879 non-null  object
 2   fnlwgt          48842 non-null  int64 
 3   education       48842 non-null  object
 4   education-num   48842 non-null  int64 
 5   marital-status  48842 non-null  object
 6   occupation      47876 non-null  object
 7   relationship    48842 non-null  object
 8   race            48842 non-null  object
 9   sex             48842 non-null  object
 10  capital-gain    48842 non-null  int64 
 11  capital-loss    48842 non-null  int64 
 12  hours-per-week  48842 non-null  int64 
 13  native-country  48568 non-null  object
dtypes: int64(6), object(8)
memory usage: 5.2+ MB
```

```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
```

```
 ---  ------  --------------  -----
  0   income  48842 non-null  object
dtypes: object(1)
memory usage: 381.7+ KB
```

Seperating Categorical and Numerical Variables

```
numerical_features = X.select_dtypes(include=['float64', 'int64'])
categorical_features = X.select_dtypes(include=['object'])

print("Numerical Features:")
print(numerical_features.columns)

print("\nCategorical Features:")
print(categorical_features.columns)
```

```
Numerical Features:
Index(['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss',
       'hours-per-week'],
      dtype='object')

Categorical Features:
Index(['workclass', 'education', 'marital-status', 'occupation',
       'relationship', 'race', 'sex', 'native-country'],
      dtype='object')
```

Handling Categorical Variable

```
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()
X['workclass'] = label_encoder.fit_transform(X['workclass'])
X['education'] = label_encoder.fit_transform(X['education'])
X['marital-status'] = label_encoder.fit_transform(X['marital-status'])
X['occupation'] = label_encoder.fit_transform(X['occupation'])
X['relationship'] = label_encoder.fit_transform(X['relationship'])
X['race'] = label_encoder.fit_transform(X['race'])
X['sex'] = label_encoder.fit_transform(X['sex'])
X['native-country'] = label_encoder.fit_transform(X['native-country'])
X.head(20)
```

```
<ipython-input-11-9f5d11bed876>:3: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['workclass'] = label_encoder.fit_transfor
<ipython-input-11-9f5d11bed876>:4: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['education'] = label_encoder.fit_transfor
<ipython-input-11-9f5d11bed876>:5: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['marital-status'] = label_encoder.fit_tra
<ipython-input-11-9f5d11bed876>:6: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['occupation'] = label_encoder.fit_transfo
<ipython-input-11-9f5d11bed876>:7: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['relationship'] = label_encoder.fit_trans
<ipython-input-11-9f5d11bed876>:8: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['race'] = label_encoder.fit_transform(X['
<ipython-input-11-9f5d11bed876>:9: SettingWit
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['sex'] = label_encoder.fit_transform(X['s
<ipython-input-11-9f5d11bed876>:10: SettingWi
A value is trying to be set on a copy of a sl
Try using .loc[row_indexer,col_indexer] = val

See the caveats in the documentation: https:/
  X['native-country'] = label_encoder.fit_tra
```

	age	workclass	fnlwgt	education	educat
0	39	7	77516	9	
1	50	6	83311	9	
2	38	4	215646	11	

3	53	4	234721	1
4	28	4	338409	9
5	37	4	284582	12
6	49	4	160187	6
7	52	6	209642	11
8	31	4	45781	12
9	42	4	159449	9
10	37	4	280464	15
11	30	7	141297	9
12	23	4	122272	9
13	32	4	205019	7
14	40	4	121772	8
15	34	4	245487	5
16	25	6	176756	11
17	32	4	186824	11
18	38	4	28887	1

Next steps: View recommended plots

```
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()
y['income'] = label_encoder.fit_transform(y['income'])
```

```
<ipython-input-12-a399ca7721c1>:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-c
  y['income'] = label_encoder.fit_transform(y['income'])
```

Remarks:

There's 9 categorical values which are 'workclass', 'education', 'marital-status', 'occupation','relationship', 'race', 'sex', 'native-country' and 'income'. I used label encoder to convert it into string since neural networks is crucial for enabling models to effectively learn from non-numerical data, improve performance, and avoid bias. So proper encoding and representation of categorical variables allow neural networks to leverage the full potential of the available data and make more accurate predictions

Verifying the conversion of Categorical into Numerical Features

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             48842 non-null  int64
 1   workclass       48842 non-null  int64
 2   fnlwgt          48842 non-null  int64
 3   education       48842 non-null  int64
 4   education-num   48842 non-null  int64
 5   marital-status  48842 non-null  int64
 6   occupation      48842 non-null  int64
 7   relationship    48842 non-null  int64
 8   race            48842 non-null  int64
 9   sex             48842 non-null  int64
 10  capital-gain    48842 non-null  int64
 11  capital-loss    48842 non-null  int64
 12  hours-per-week  48842 non-null  int64
 13  native-country  48842 non-null  int64
dtypes: int64(14)
memory usage: 5.2 MB
```

```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   income  48842 non-null  int64
dtypes: int64(1)
memory usage: 381.7 KB
```

Handling Missing Values

```
X.isnull().sum()
```

```
age               0
workclass         0
fnlwgt            0
education         0
```

```
education-num     0
marital-status    0
occupation        0
relationship      0
race              0
sex               0
capital-gain      0
capital-loss      0
hours-per-week    0
native-country    0
dtype: int64
```

```
y.isnull()
```

	income
0	False
1	False
2	False
3	False
4	False
...	...
48837	False
48838	False
48839	False
48840	False
48841	False

48842 rows × 1 columns

```
X["workclass"].fillna(X["workclass"].mean(), inplace=True)
X["occupation"].fillna(X["occupation"].mean(), inplace=True)
X["native-country"].fillna(X["native-country"].mean(), inplace=True)
```

```
<ipython-input-17-bcb232619632>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-c
  X["workclass"].fillna(X["workclass"].mean(), inplace=True)
<ipython-input-17-bcb232619632>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-c
  X["occupation"].fillna(X["occupation"].mean(), inplace=True)
<ipython-input-17-bcb232619632>:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-c
  X["native-country"].fillna(X["native-country"].mean(), inplace=True)
```

Verifying the missing values

```
X.isnull().sum()
```

```
age               0
workclass         0
fnlwgt            0
education         0
education-num     0
marital-status    0
occupation        0
relationship      0
race              0
sex               0
capital-gain      0
capital-loss      0
hours-per-week    0
native-country    0
dtype: int64
```

Remarks for missing values

After the label being done converted the categorical transform into numerical datatype. All of the variables there are now ready to handle any pattern or model since there all in numerical form

```
X.describe()
```

	age	workclass	fnlwg
count	48842.000000	48842.000000	4.884200e+(
mean	38.643585	4.047889	1.896641e+(
std	13.710510	1.528374	1.056040e+(
min	17.000000	0.000000	1.228500e+(
25%	28.000000	4.000000	1.175505e+(
50%	37.000000	4.000000	1.781445e+(
75%	48.000000	4.000000	2.376420e+(

Remarks for describe function

The result above, regarding with the 25% and 75% is a big gap compared to each of them respectively. This proves that there's an outliers goes within the variables, to visualized and to verify those things a box plot will be performed

Handling Outliers

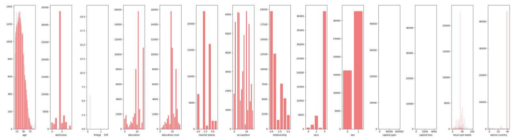
```
X.columns, y.columns
```

```
(Index(['age', 'workclass', 'fnlwgt', 'education', 'education-num',
        'marital-status', 'occupation', 'relationship', 'race', 'sex',
        'capital-gain', 'capital-loss', 'hours-per-week', 'native-
country'],
       dtype='object'),
 Index(['income'], dtype='object'))
```

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

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

```
fig, axs = plt.subplots(1, len(columns), figsize=(30, 8))

for i, col in enumerate(columns):
    value_counts = X[col].value_counts()
    axs[i].bar(value_counts.index, value_counts.values, color=('lightcoral')
    axs[i].set_xlabel(col)

plt.tight_layout()
plt.show()
```



```
X.shape, y.shape
```

```
((48842, 14), (48842, 1))
```

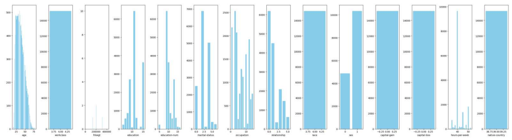
Remarks of Detecting Outliers

As you can see there's 48,842 instances that have been detected.

Removing Outliers using IQR

Trimming, or truncating, is the process of removing observations that show outliers in one or more variables in the dataset. There are three commonly used methods to set the boundaries beyond which a value can be considered an outlier

```
def IQR_outliers(df, column_name, thresh=1.5):
  Q1 = df[column_name].quantile(0.25)
  Q3 = df[column_name].quantile(0.75)
  IQR = Q3 - Q1
  lower_bound = Q1 - (thresh * IQR)
  upper_bound = Q3 + (thresh * IQR)

  return df[(df[column_name] >= lower_bound) & (df [column_name] <= upper_bc


indices_to_keep = np.array([])
for col in columns:
    outliers_removed_X = IQR_outliers(X, col)
    indices_to_keep = np.intersect1d(indices_to_keep, outliers_removed_X.inc

X = X.loc[indices_to_keep]
y = y.loc[indices_to_keep]


fig, axs = plt.subplots(1,len(columns), figsize=(30, 8))

for i, col in enumerate(columns):
    value_counts = X[col].value_counts()
    axs[i].bar(value_counts.index, value_counts.values, color='skyblue')
    axs[i].set_xlabel(col)

plt.tight_layout()
plt.show()
```



```
X.shape, y.shape
```

```
((15221, 14), (15221, 1))
```

Verifying after Removing Outliers

Using IQR Rule the instances are now 15,221 and 27,621 instances have been removed

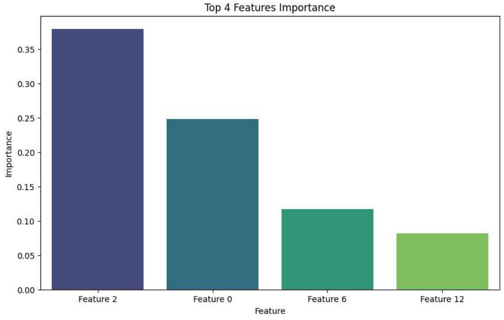
Standardization of Data

```
from sklearn.preprocessing import StandardScaler

normalize = StandardScaler()
X_norm = normalize.fit_transform(X)
```

Feature Selection

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_norm, y)

importances = clf.feature_importances_
indices = np.argsort(importances)[::-1]

k = 4
top_k_features = indices[:k]
top_features_df = pd.DataFrame({'Feature': [f'Feature {i}' for i in top_k_fea
                                'Importance': importances[top_k_features]})

plt.figure(figsize=(10, 6))
sns.barplot(data=top_features_df, x='Feature', y='Importance', palette='virid
plt.title(f'Top {k} Features Importance')
plt.xlabel('Feature')
plt.ylabel('Importance')
plt.show()
```

```
<ipython-input-29-5ef8c1877652>:4: DataConver
  clf.fit(X_norm, y)
<ipython-input-29-5ef8c1877652>:15: FutureWar

Passing `palette` without assigning `hue` is

  sns.barplot(data=top_features_df, x='Featur
```




```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 15221 entries, 2 to 48839
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             15221 non-null  int64
 1   workclass       15221 non-null  int64
 2   fnlwgt          15221 non-null  int64
 3   education       15221 non-null  int64
 4   education-num   15221 non-null  int64
 5   marital-status  15221 non-null  int64
 6   occupation      15221 non-null  int64
 7   relationship    15221 non-null  int64
 8   race            15221 non-null  int64
 9   sex             15221 non-null  int64
```

```
 10  capital-gain    15221 non-null  int64
 11  capital-loss    15221 non-null  int64
 12  hours-per-week  15221 non-null  int64
 13  native-country  15221 non-null  int64
dtypes: int64(14)
memory usage: 1.7 MB
```

Remarks for Feature Selection

The result was there are 4 most important feature variable here such as fnlwgt, age, occupation, and hours-per-week which may be a factor for predicting the exact income level of an individual based on those features. This one is good for classification problems whether the person's income can actually exceeds to 50,000 or not or for regression problem to predict the exact income level of these people.

```
X = X_norm[:,[2,0,6,12]]

print(X)
```

```
[[ 4.40332349e-01  8.79850060e-02 -1.54653857e-01 -4.21795495e-01]
 [ 1.24967974e+00  1.35457865e-03 -6.36041661e-01 -4.21795495e-01]
 [ 1.01945868e-01 -4.31797558e-01  8.60400447e-02 -4.21795495e-01]
 ...
 [ 7.87442068e-01 -3.45167131e-01  8.08121750e-01 -4.21795495e-01]
 [ 4.37667242e-01  1.74615433e-01  8.08121750e-01 -1.42294932e+00]
 [ 2.31103830e+00  8.79850060e-02  8.08121750e-01  2.08108908e+00]]
```

Creating a Model

```
from keras.models  import Sequential
from keras.layers import Input, Dense
from keras.optimizers import SGD


model = Sequential([Dense(20, activation='relu', input_shape=(X.shape[1],)),
                    Dense(20, activation='relu'),
                    Dense(1, activation='sigmoid')])

model.compile(optimizer=SGD(learning_rate=0.009),
              loss='binary_crossentropy',
              metrics=['accuracy'])

print(f"Accuracy: {model.evaluate(X, y, verbose=0)[1]*100:.2f}%")
```

```
Accuracy: 51.87%
```

T1. Saving the model in HDF5 format

```
!pip install h5py
```

```
Requirement already satisfied: h5py in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.1
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount,
```

```
model.save("/content/drive/MyDrive/CPE018 - A8.1 / model.h5")
print("Saved the model weights in HDF5 format to Google Drive.")
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:31€
  saving_api.save_model(
Saved the model weights in HDF5 format to Google Drive.
```

Remarks for HDF5 Format

It's efficient, versatile data storage format widely used in scientific computing and machine learning for its speed, compression, and support for hierarchical data structures.

T2. Save a model and load in JSON Format

```
from tensorflow.keras.models import model_from_json

model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

model.save("/content/drive/MyDrive/CPE018 - A8.1 / model_json.h5")
print("Saved the model in JSON format to Google Drive.")
```

```
Saved the model in JSON format to Google Drive.
```

```
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)

loaded_model.load_weights("/content/drive/MyDrive/CPE018 - A8.1 / model_jsor
print("Loaded model from disk")
```

```
Loaded model from disk
```

```
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=
score = loaded_model.evaluate(X, y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

```
accuracy: 51.87%
```

Remarks for JSON Format

JSON's broader compatibility and adoption make it a preferred choice for many applications.

T3. Save a model and load in YAML Format

```
model_yaml = model.to_json()
with open("model.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)

model.save("/content/drive/MyDrive/CPE018 - A8.1 / model_yaml.h5")
print("Saved the model in YAML format to Google Drive.")
```

```
Saved the model in YAML format to Google Drive.
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:31(
  saving_api.save_model(
```

```
yaml_file = open('model.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
loaded_model = model_from_json(loaded_model_yaml)

loaded_model.load_weights("/content/drive/MyDrive/CPE018 - A8.1 / model_yaml
print("Loaded model from disk")
```

```
Loaded model from disk
```

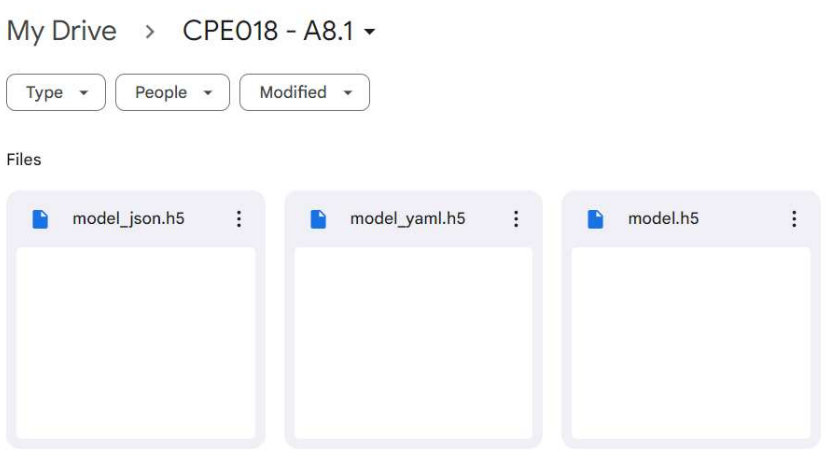
```
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metric
score = loaded_model.evaluate(X, y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

```
accuracy: 51.87%
```

Remarks for YAML Format

YAML and JSON are both data formats used for storing and exchanging information. YAML is prized for its human-readable structure, using indentation for organization, which makes it easy to understand and write. JSON, on the other hand, is widely supported across different platforms and languages, making it popular for web development and APIs. While YAML is simpler and more intuitive for humans, JSON's broader compatibility and adoption make it a preferred choice for many applications.

Verifying the loaded model from Disk



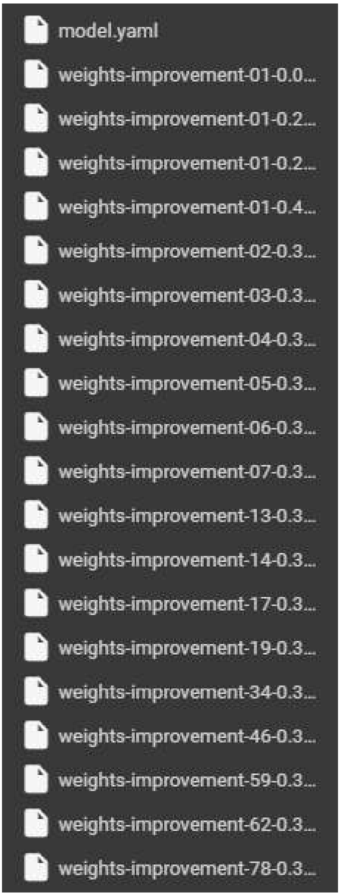
T4. Checkpoint Neural Network Model Improvements

```
from keras.callbacks import ModelCheckpoint

filepath = "weights-improvement-{epoch:02d}-{val_accuracy:.2f}.keras"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, sav
```

```
callbacks_list = [checkpoint]

model.fit(X, y, validation_split=0.5, epochs=100, batch_size=20, callbacks=ca
```

```
Epoch 94: val_accuracy did not improve from 0.37722

Epoch 95: val_accuracy improved from 0.37722 to 0.42228, saving model
```

Verifying the Model Improvement



Reamarks for Model Improvement

As you can see above it only save the a checkpoint if there's an improvement in model's perfomances.

T5. Checkpoint Best Neural Network Model only

```
filepath= "weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, sa
callbacks_list = [checkpoint]


model.fit(X, y, validation_split=0.5, epochs=100, batch_size=20, callbacks=c
```

```
Epoch 93: val_accuracy did not improve from 0.44396

Epoch 94: val_accuracy did not improve from 0.44396

Epoch 95: val_accuracy did not improve from 0.44396

Epoch 96: val accuracy did not improve from 0.44396
```

weights.best.hdf5

Remarks

It only save the best weights that was generated in HDF5 format. This is very useful to practice experimenting the accuracy of the model.

T6. Load a saved Neural Network Model

```
model = Sequential()
model.add(Dense(20, input_dim=4, kernel_initializer = 'uniform' , activatior
model.add(Dense(20, kernel_initializer= 'uniform' , activation= 'relu' ))
model.add(Dense(1, kernel_initializer= 'uniform' , activation= 'sigmoid' ))

model.load_weights("weights.best.hdf5")

model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=['ac
print("Created model and loaded weights from file")
```

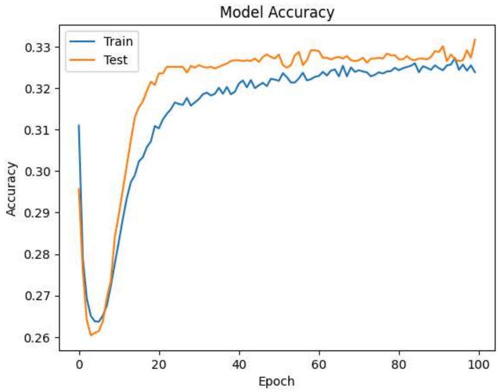
```
Created model and loaded weights from file
```

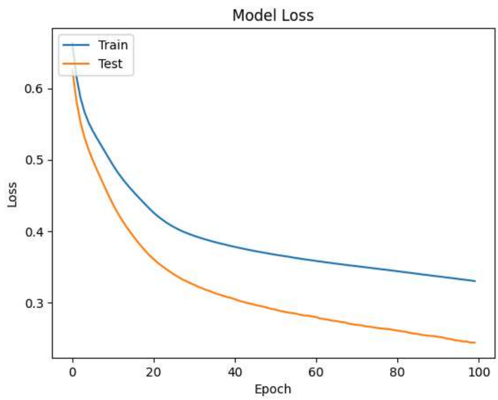
Remarks for Saving Neural Network Model

As you can see above, it was successfully create a model and loaded weights from the file. It demonstrate of giving an access to best weights that's being generated above.

T7. Visualized Model Training History in Keras

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
import matplotlib.pyplot as plt


np.random.seed(10)
X_train1, X_test1, y_train1, y_test1 = train_test_split(X, y, test_size=0.33


model1 = Sequential([
            Dense(3, activation='relu', input_shape=(X_train1.shape[1],)),
            Dense(2, activation='relu'),
            Dense(1, activation='sigmoid')
        ])

optimizer = SGD(learning_rate=0.009)
model1.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['ac

history = model1.fit(X_train1, y_train1, validation_data=(X_test1, y_test1),


plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```


Model Accuracy
Train
Test
Accuracy
Epoch
0.33
0.32
0.31
0.30
0.29
0.28
0.27
0.26
0
20
40
60
80
100


Model Loss
Train
Test
Loss
Epoch
0.6
0.5
0.4
0.3
0
20
40
60
80
100

T8. Show the applicaiton of Droupout Regularization

A simple hidden layer dropout was added in the model that was created.

```
np.random.seed(10)
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, test_size=0.5,
```

```
from tensorflow.keras.layers import Dense, Dropout

model2 = Sequential([
            Dense(50, activation='relu', input_shape=(4,)),
            Dropout(0.5),
            Dense(50, activation='relu'),
            Dropout(0.5),
            Dense(1, activation='sigmoid')
        ])

optimizer = SGD(learning_rate = 0.05)
model2.compile(optimizer= optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])

history2 = model2.fit(X_train2, y_train2, validation_data=(X_test2,y_test2),
```

```
train_loss, train_accuracy = model2.evaluate(X_train2, y_train2, verbose=0)
test_loss, test_accuracy = model2.evaluate(X_test2, y_test2, verbose=0)
print(f"Training Accuracy: {train_accuracy:.4f}, Training Loss: {train_loss:
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")
```

```
Training Accuracy: 0.5242, Training Loss: nan
Test Accuracy: 0.5133, Test Loss: nan
```

Remarks for Dropout Regularization

Dropout in neural networks is used to prevent over-reliance on certain neurons. It randomly turns off some neurons during training, forcing the network to learn more robust features. It helps me to prevent overfitting and improves the model's ability to generalize to new, unseen data.

T9. Show the application of Dropout on the visible layer

```
from keras.optimizers import SGD
from keras.layers import Dropout

np.random.seed(10)
model3 = Sequential([
            Dropout(0.8, input_shape=(4,)),
            Dense(5, activation='relu'),
            Dense(4, activation='relu'),
            Dense(1, activation='sigmoid')
        ])

optimizer = SGD(learning_rate = 0.05,momentum = 0.9)

model3.compile(optimizer= optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])

history3 = model3.fit(X_train2, y_train2, validation_data=(X_test2,y_test2),

train_loss, train_accuracy = model3.evaluate(X_train2, y_train2, verbose=0)
test_loss, test_accuracy = model3.evaluate(X_test2, y_test2, verbose=0)
print(f"Training Accuracy: {train_accuracy:.4f}, Training Loss: {train_loss:
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")
```

```
Training Accuracy: 0.2625, Training Loss: 0.4945
Test Accuracy: 0.2562, Test Loss: 0.4289
```

Remarks for Dropout on visible layer

it encourages the model to learn more diverse patterns and reduces the risk of overfitting. This improves the model's ability to generalize to new data and enhances its overall performance.

T10. Show the application of Droupout on the hiddent layer

```
from keras.optimizers import SGD
from keras.layers import Dropout

np.random.seed(10)
model4 = Sequential([
            Dense(4, activation='relu', input_shape=(4,)),
            Dense(3, activation='relu'),
            Dropout(0.05),
            Dense(1, activation='sigmoid')
        ])

optimizer = SGD(learning_rate = 0.33,momentum = 0.9)

model4.compile(optimizer= optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])

history4 = model4.fit(X_train2, y_train2, validation_data=(X_test2,y_test2),

train_loss, train_accuracy = model4.evaluate(X_train2, y_train2, verbose=0)
test_loss, test_accuracy = model4.evaluate(X_test2, y_test2, verbose=0)
print(f"Training Accuracy: {train_accuracy:.4f}, Training Loss: {train_loss:
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")
```

```
Training Accuracy: 0.2625, Training Loss: 0.5976
Test Accuracy: 0.2562, Test Loss: 0.5326
```

Remarks for Dropout the hidden layer

The resulted to the comparison of the best model's performance to others wasn't good enough but it depends ob the dataset you're relying on

11. Show the application of a time-based learning rate schedule

```
from tensorflow.keras.optimizers.legacy import SGD
from keras.layers import Dropout

model5 = Sequential([
            Dense(4, activation='relu', input_shape=(4,)),
            Dense(3, activation='relu'),
            Dropout(0.05),
            Dense(1, activation='sigmoid')
        ])

epochs = 100
learning_rate = 0.09
```

```
decay_rate = learning_rate / epochs
momentum = 0.9
optimizer =  SGD(learning_rate=learning_rate, momentum=momentum, decay=decay_

model5.compile(optimizer= optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])

history5 = model5.fit(X_train2, y_train2, validation_data=(X_test2,y_test2), 
```

...

```
Epoch 91/100
238/238 - 1s - loss: 0.5522 - accuracy: 0.2625 - val_loss: 0.5131 - v
Epoch 92/100
238/238 - 1s - loss: 0.5523 - accuracy: 0.2625 - val_loss: 0.5125 - v
Epoch 93/100
238/238 - 1s - loss: 0.5525 - accuracy: 0.2625 - val_loss: 0.5128 - v
Epoch 94/100
238/238 - 1s - loss: 0.5523 - accuracy: 0.2625 - val_loss: 0.5112 - v
Epoch 95/100
238/238 - 1s - loss: 0.5525 - accuracy: 0.2625 - val_loss: 0.5121 - v
```

Remarks for time-based learning rate

The time-based learning rate schedule, also known as the decay schedule, adjusts the learning rate at each epoch automatically. This automated adjustment helps fine-tune the model efficiently without the need for manual intervention in every iteration.

12. Show the application of a drop-based learning rate schedule

```
from tensorflow.keras.optimizers.legacy import SGD
from keras.layers import Dropout
from tensorflow.keras.callbacks import LearningRateScheduler
import math
```

```
def step_decay(epoch):
    initial_lrate = 0.1
    drop = 0.5
    epochs_drop = 10.0
```