INDEX

S.No	Name of the Experiment	Date	Sign
1.	Build an Artificial Neural Network to implement Regression task using the Back-propagation algorithm and test the same using appropriate data sets	15/12/23	
2.	Build an Artificial Neural Network to implement Binary Classification task using the Back-propagation algorithm and test the same using appropriate data sets.	8/01/24	
3.	Build an Artificial Neural Network to implement Multi-Class Classification task using the Back-propagation algorithm and test the same using appropriate data sets.	15/01/24	
4.	Implement an image classification task using pre-trained models like AlexNet, VGGNet, InceptionNet and ResNet and compare the results.	29/01/24	
5.	Design a CNN architecture to implement the image classification task over an image dataset. Perform the Hyper-parameter tuning and record the results.	5/02/24	
6.	Implement a deep neural network (Ex: UNet / SegNet) for Image Segmentation task.	10/02/24	
7.	Train a Recurrent Neural Network (RNN) on the IMDB large movie review dataset for sentiment analysis.	18/02.24	
8.	Download the tweets data from twitter and run an RNN/LSTM to classify the sentiment as Hate/ Non-hate speech. Compare the results of both the models.	4/03/24	
9.	Implement GAN architecture on MNIST dataset to recognize the handwritten digits.	11/03/24	

Experiment:- 1

Aim:- Build an Artificial Neural Network to implement Regression task using the Back-propagation algorithm and test the same using appropriate data sets.

Importing Necessary Libraries

```
import numpy as np
import pandas as pd
```

Loading the Churn Dataset

This data set contains details of a bank's customers and the target variable is a binary variable reflecting the fact whether the customer left the bank (closed his account) or he continues to be a customer.

Binary flag 1 if the customer closed account with bank and 0 if the customer is retained.

```
churn_data = pd.read_csv('Churn_Modelling.csv', delimiter = ',')
churn_data.head(5)
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balanc
0	1	15634602	Hargrave	619	France	Female	42	2	0.0
1	2	15647311	Hill	608	Spain	Female	41	1	83807.8
2	3	15619304	Onio	502	France	Female	42	8	159660.8
3	4	15701354	Boni	699	France	Female	39	1	0.0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.8

Accessing the Column Names in the Dataset

```
churn_data.columns
```

```
Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
       'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
       'IsActiveMember', 'EstimatedSalary', 'Exited'],
      dtype='object')
```

Setting Column as a Index

```
churn_data = churn_data.set_index('RowNumber')
churn_data.head()
```

RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance
1	15634602	Hargrave	619	France	Female	42	2	0.00
2	15647311	Hill	608	Spain	Female	41	1	83807.86
3	15619304	Onio	502	France	Female	42	8	159660.80
4	15701354	Boni	699	France	Female	39	1	0.00
5	15737888	Mitchell	850	Spain	Female	43	2	125510.82

Finding the Shape of the Dataset

```
churn_data.shape
```

```
(10000, 13)
```

```
churn_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 1 to 10000
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   CustomerId       10000 non-null  int64  
 1   Surname          10000 non-null  object 
 2   CreditScore      10000 non-null  int64  
 3   Geography        10000 non-null  object 
 4   Gender           10000 non-null  object 
 5   Age              10000 non-null  int64  
 6   Tenure           10000 non-null  int64  
 7   Balance          10000 non-null  float64
 8   NumOfProducts    10000 non-null  int64  
 9   HasCrCard        10000 non-null  int64  
 10  IsActiveMember   10000 non-null  int64  
 11  EstimatedSalary  10000 non-null  float64
 12  Exited           10000 non-null  int64  
dtypes: float64(2), int64(8), object(3)
memory usage: 1.1+ MB
```

Checking Missing Values

```
churn_data.isna().sum()
```

```
CustomerId         0
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

Some Columns are Totally Unproductive so let's Remove them

```
churn_data.nunique()
```

```
CustomerId         10000
Surname             2932
CreditScore          460
Geography              3
Gender                 2
Age                   70
Tenure                11
Balance             6382
NumOfProducts          4
HasCrCard              2
IsActiveMember         2
EstimatedSalary     9999
Exited                 2
dtype: int64
```

```
churn_data.drop(['CustomerId','Surname'],axis=1,inplace=True)
```

```
churn_data.head()
```

RowNumber	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
1	619	France	Female	42	2	0.00	1	1	1	101348.88	1
2	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
3	502	France	Female	42	8	159660.80	3	1	0	113931.57	1
4	699	France	Female	39	1	0.00	2	0	0	93826.63	0
5	850	Spain	Female	43	2	125510.82	1	1	1	79084.10	0

```
churn_data.shape
```

```
(10000, 11)
```

Some Visualizations

[ ] ↳ *7 cells hidden*

Label Encoding of Categorical Variables

Label Encoding means converting categorical features into numerical values. So that they can be fitted by machine learning models which only take numerical data.

Example: Suppose we have a column Height in some dataset that has elements as Tall, Medium, and short. To convert this categorical column into a numerical column we will apply label encoding to this column. After applying label encoding, the Height column is converted into a numerical column having elements 0,1, and 2 where 0 is the label for tall, 1 is the label for medium, and 2 is the label for short height.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
churn_data[['Geography', 'Gender']] = churn_data[['Geography', 'Gender']].apply(le.fit_transform)
```

```
churn_data.head()
```

RowNumber	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
1	619	0	0	42	2	0.00	1	1	1	101348.88	1
2	608	2	0	41	1	83807.86	1	0	1	112542.58	0
3	502	0	0	42	8	159660.80	3	1	0	113931.57	1
4	699	0	0	39	1	0.00	2	0	0	93826.63	0
5	850	2	0	43	2	125510.82	1	1	1	79084.10	0

Seperating Label from Data

```
y = churn_data.Exited
X = churn_data.drop(['Exited'],axis=1)
```

```
X.columns
```

```
Index(['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance',
       'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'EstimatedSalary'],
      dtype='object')
```

```
y
```

```
RowNumber
1        1
2        0
3        1
```

```
4       0
5       0
       ..
9996    0
9997    0
9998    1
9999    1
10000   0
Name: Exited, Length: 10000, dtype: int64
```

Splitting the Data into Training and Testing

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 2)
```

```
print("Shape of the X_train", X_train.shape)
print("Shape of the X_test", X_test.shape)
print("Shape of the y_train", y_train.shape)
print("Shape of the y_test", y_test.shape)
```

```
Shape of the X_train (7000, 10)
Shape of the X_test (3000, 10)
Shape of the y_train (7000,)
Shape of the y_test (3000,)
```

Feature Scaling:

The result of **standardization** (or **Z-Score normalization**) is that the features will be re scaled so that they'll have the properties of a standard normal distribution with:

$$\mu = 0$$

And

$$\sigma = 1$$

Where μ is the mean(average) and σ is the standard deviation from the mean; standard scores (also called **Z** scores) of the sampels are calculated as follows:

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

About Min-Max Scaling

An alternative approach to **Z-Score** normalization (or called standardization) is the so-called **Min-Max Scaling** (often also simply called **Normalization** - a common cause for ambiguities)

In this approach, the data is scaled to a fixed range - usually `[0, 1]`. The cost of having this bounded range - in contrast to standrdization - is that we will end up with smaaller standard deviations, which can suppress the effect of outliers.

Note:

If the dataset have lot's of outliers, and the algorithms are sensitive to outliers, please use `Min-Max Scaler`

A `Min-Max Scaling` is typically done via the foloowing equation:

$$X_{norm} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

X_i is the i^{th} sample of dataset.

Z-Score Standardization or Min-Max Scaling

"Standardization or Min-Max scaling"? - There is no obvious answer to this question: it really depends on the application.

However this doesn't mean that `Min-Max Scaling` is not useful at all, A popular application is `image processing`, where pixel intensities have to be normalized to fit withint a certain range (i.e., `[0, 255]` for the RGB colour range). Also, typical *Neural Network* Algorithm require data that on `- 1` scale.

```
a 0
```

Need for Normalization

For example, consider a data set containing two features, age(x1), and income(x2). Where age ranges from 0–100, while income ranges from 0–20,000 and higher. Income is about 1,000 times larger than age and ranges from 20,000–500,000. So, these two features are in very different ranges. When we do further analysis, like multivariate linear regression, for example, the attributed income will intrinsically influence the result more due to its larger value. But this doesn't necessarily mean it is more important as a predictor.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Building the ANN Model

```
# sequential model to initialise our ann and dense module to build the layers
from keras.models import Sequential
from keras.layers import Dense
```

```
classifier = Sequential()
# Adding the input layer and the first hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activation = 'relu', input_dim = 10))

# Adding the second hidden layer
classifier.add(Dense(units = 16, kernel_initializer = 'uniform', activation = 'relu'))

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
```

Compiling and Fitting the Model

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])

# Fitting the ANN to the Training set
classifier.fit(X_train, y_train, batch_size = 10, epochs = 100, verbose = 1)
```

```
Epoch 78/100
700/700 [==============================] - 1s 805us/step - loss: 0.3867 - accuracy: 0.8400
Epoch 79/100
700/700 [==============================] - 1s 805us/step - loss: 0.3865 - accuracy: 0.8401
Epoch 80/100
700/700 [==============================] - 1s 872us/step - loss: 0.3868 - accuracy: 0.8411
Epoch 81/100
700/700 [==============================] - 1s 827us/step - loss: 0.3860 - accuracy: 0.8404
Epoch 82/100
700/700 [==============================] - 1s 827us/step - loss: 0.3857 - accuracy: 0.8384
Epoch 83/100
700/700 [==============================] - 1s 805us/step - loss: 0.3855 - accuracy: 0.8417
Epoch 84/100
700/700 [==============================] - 1s 760us/step - loss: 0.3858 - accuracy: 0.8407
Epoch 85/100
700/700 [==============================] - 1s 782us/step - loss: 0.3859 - accuracy: 0.8424
Epoch 86/100
700/700 [==============================] - 1s 782us/step - loss: 0.3849 - accuracy: 0.8404
Epoch 87/100
700/700 [==============================] - 1s 805us/step - loss: 0.3839 - accuracy: 0.8413
Epoch 88/100
700/700 [==============================] - 1s 760us/step - loss: 0.3846 - accuracy: 0.8400
Epoch 89/100
```

Testing the Model

```
score, acc = classifier.evaluate(X_train, y_train,
                            batch_size=10)
print('Train score:', score)
print('Train accuracy:', acc)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.5)

print('*'*20)
score, acc = classifier.evaluate(X_test, y_test,
                            batch_size=10)
print('Test score:', score)
print('Test accuracy:', acc)
```

```
700/700 [==============================] - 1s 782us/step - loss: 0.3342 - accuracy: 0.8631
Train score: 0.33418112993240356
Train accuracy: 0.8631428480148315
********************
300/300 [==============================] - 0s 784us/step - loss: 0.3537 - accuracy: 0.8550
Test score: 0.35368815064430237
Test accuracy: 0.8550000190734863
```

Confusion Matrix

* Accuracy

number of examples correctly predicted / total number of examples
$ACC = (TP + TN)/(TP + FP + FN + TN)$

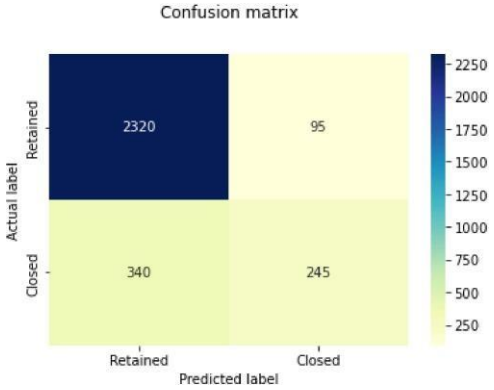
```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
target_names = ['Retained', 'Closed']
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[2320   95]
 [ 340  245]]
```

```
import matplotlib.pyplot as  plt
import seaborn as sns
```

```
p = sns.heatmap(pd.DataFrame(cm), annot=True, xticklabels=target_names, yticklabels=target_names, cmap="YlGnBu" ,fmt='g')
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

```
Text(0.5, 15.0, 'Predicted label')
```



Classification Report

* True Positive Rate

number of samples actually and predicted as `Positive` / total number of samples actually `Positive`
Also called **Sensitivity or Recall**.
$$TPR = TP/P = TP/(TP + FN)$$

* Positive Predictive Value

number of samples actually and predicted as `Positive` / total number of samples predicted as `Positive`
Also called **Precision**.
$$PPV = TP/(TP + FP)$$

* F1 score

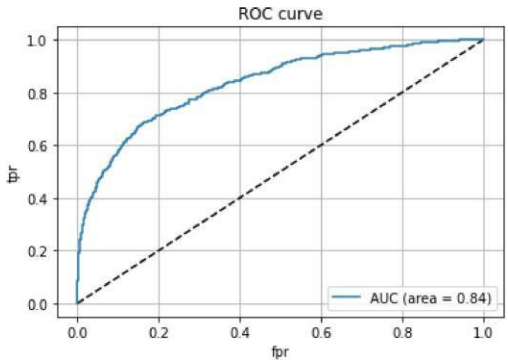
Harmonic Mean of Precision and Recall.
$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

```
#import classification_report
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred, target_names=target_names))
```

```
                precision    recall  f1-score   support

      Retained       0.87      0.96      0.91      2415
        Closed       0.72      0.42      0.53       585

      accuracy                           0.85      3000
     macro avg       0.80      0.69      0.72      3000
  weighted avg       0.84      0.85      0.84      3000
```

ROC curve

```
from sklearn.metrics import roc_curve, auc
y_pred_proba = classifier.predict(X_test)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='AUC (area = %0.2f)' % roc_auc)
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.grid()
plt.legend(loc="lower right")
plt.title('ROC curve')
plt.show()
```




```
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred_proba)
```

```
0.8360007786094743
```

Finetuing the Network

```
# Tuning the ANN
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
def build_classifier(optimizer):
    classifier = Sequential()
    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation = 'relu', input_dim = 10))
    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
    classifier.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = KerasClassifier(build_fn = build_classifier)
parameters = {'batch_size': [16, 32],
              'epochs': [50, 100],
              'optimizer': ['adam', 'rmsprop']}
grid_search = GridSearchCV(estimator = classifier,
                           param_grid = parameters,
                           scoring = 'accuracy', cv = 2) #,cv = 10
grid_search = grid_search.fit(X_train, y_train,verbose = 1)
best_parameters = grid_search.best_params_
best_accuracy = grid_search.best_score_
```

```
Epoch 26/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4165 - accuracy: 0.8311
Epoch 27/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4159 - accuracy: 0.8320
Epoch 28/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4153 - accuracy: 0.8326
Epoch 29/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4146 - accuracy: 0.8326
Epoch 30/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4139 - accuracy: 0.8326
Epoch 31/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4134 - accuracy: 0.8349
Epoch 32/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4128 - accuracy: 0.8351
Epoch 33/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4123 - accuracy: 0.8360
Epoch 34/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4119 - accuracy: 0.8340
Epoch 35/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4115 - accuracy: 0.8360
Epoch 36/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4109 - accuracy: 0.8357
Epoch 37/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4106 - accuracy: 0.8380
Epoch 38/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4103 - accuracy: 0.8366
Epoch 39/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4098 - accuracy: 0.8371
Epoch 40/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4094 - accuracy: 0.8360
Epoch 41/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4090 - accuracy: 0.8380
Epoch 42/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4086 - accuracy: 0.8380
Epoch 43/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4083 - accuracy: 0.8377
Epoch 44/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4081 - accuracy: 0.8377
Epoch 45/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4078 - accuracy: 0.8374
Epoch 46/100
110/110 [==============================] - 0s 1ms/step - loss: 0.4075 - accuracy: 0.8389
```

```
best_parameters
```

```
{'batch_size': 16, 'epochs': 50, 'optimizer': 'rmsprop'}
```

```
best_accuracy
```

Experiment : 2

AIM: Build an Artificial Neural Network to implement Binary Classification task using the Back-propagation algorithm and test the same using appropriate data sets.

Importing Necessary Libraries

In [ ]:
```
import numpy as np
import pandas as pd
```

Loading the Churn Dataset

This data set contains details of a bank's customers and the target variable is a binary variable reflecting the fact whether the customer left the bank (closed his account) or he continues to be a customer.

Binary flag 1 if the customer closed account with bank and 0 if the customer is retained.

In [ ]:
```
churn_data = pd.read_csv('Churn_Modelling.csv', delimiter = ',')
churn_data.head(5)
```

Out[ ]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Ten
0	1	15634602	Hargrave	619	France	Female	42	
1	2	15647311	Hill	608	Spain	Female	41	
2	3	15619304	Onio	502	France	Female	42	
3	4	15701354	Boni	699	France	Female	39	
4	5	15737888	Mitchell	850	Spain	Female	43	

Accessing the Column Names in the Dataset

In [ ]:
```
churn_data.columns
```

Out[ ]:
```
Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
       'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCar
d',
       'IsActiveMember', 'EstimatedSalary', 'Exited'],
      dtype='object')
```

Setting Column as a Index

In [ ]:
```
churn_data = churn_data.set_index('RowNumber')
churn_data.head()
```

Finding the Shape of the Dataset

In [ ]:
```
churn_data.shape
```

Out[ ]:
```
(10000, 13)
```

In [ ]:
```
churn_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10000 entries, 1 to 10000
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   CustomerId       10000 non-null  int64  
 1   Surname          10000 non-null  object 
 2   CreditScore      10000 non-null  int64  
 3   Geography        10000 non-null  object 
 4   Gender           10000 non-null  object 
 5   Age              10000 non-null  int64  
 6   Tenure           10000 non-null  int64  
 7   Balance          10000 non-null  float64
 8   NumOfProducts    10000 non-null  int64  
 9   HasCrCard        10000 non-null  int64  
 10  IsActiveMember   10000 non-null  int64  
 11  EstimatedSalary  10000 non-null  float64
 12  Exited           10000 non-null  int64  
dtypes: float64(2), int64(8), object(3)
memory usage: 1.1+ MB
```

Checking Missing Values

In [ ]:
```
churn_data.isna().sum()
```

Out[ ]:
```
CustomerId          0
Surname             0
CreditScore         0
Geography           0
Gender              0
Age                 0
Tenure              0
Balance             0
NumOfProducts       0
HasCrCard           0
IsActiveMember      0
EstimatedSalary     0
Exited              0
dtype: int64
```

Some Columns are Totally Unproductive so let's Remove them

In [ ]:
```
churn_data.nunique()
```

Out[ ]:
```
CustomerId         10000
Surname             2932
CreditScore          460
Geography              3
Gender                 2
Age                   70
Tenure                11
Balance             6382
NumOfProducts          4
HasCrCard              2
IsActiveMember         2
EstimatedSalary     9999
Exited                 2
dtype: int64
```

In [ ]:
```
churn_data.drop(['CustomerId','Surname'],axis=1,inplace=True)
```

In [ ]:
```
churn_data.head()
```

Out[ ]:

RowNumber	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfPro
1	619	France	Female	42	2	0.00	
2	608	Spain	Female	41	1	83807.86	
3	502	France	Female	42	8	159660.80	
4	699	France	Female	39	1	0.00	
5	850	Spain	Female	43	2	125510.82	

In [ ]:
```
churn_data.shape
```

Out[ ]: (10000, 11)

Some Visualizations

In [ ]:
```
from matplotlib import pyplot as plt
import seaborn as sns
from scipy import stats
df = churn_data.copy()
```

In [ ]:
```
def plot_univariate(col):
    if(df[col].nunique()>2):
        plt.figure(figsize=(10,7))
        h = 0.15
        rot=90
    else:
        plt.figure(figsize=(6,6))
        h = 0.5
        rot=0
    plot = sns.countplot(x = df[col], palette='pastel')

    for bars in plot.containers:
        for p in bars:
            plot.annotate(format(p.get_height()), (p.get_x() + p.get_widt
                    ha = 'center', va = 'bottom')
            plot.annotate(f'{p.get_height()*100/df[col].shape[0] : .1f}%'
                    ha = 'center', va = 'bottom', rotation=rot)
```

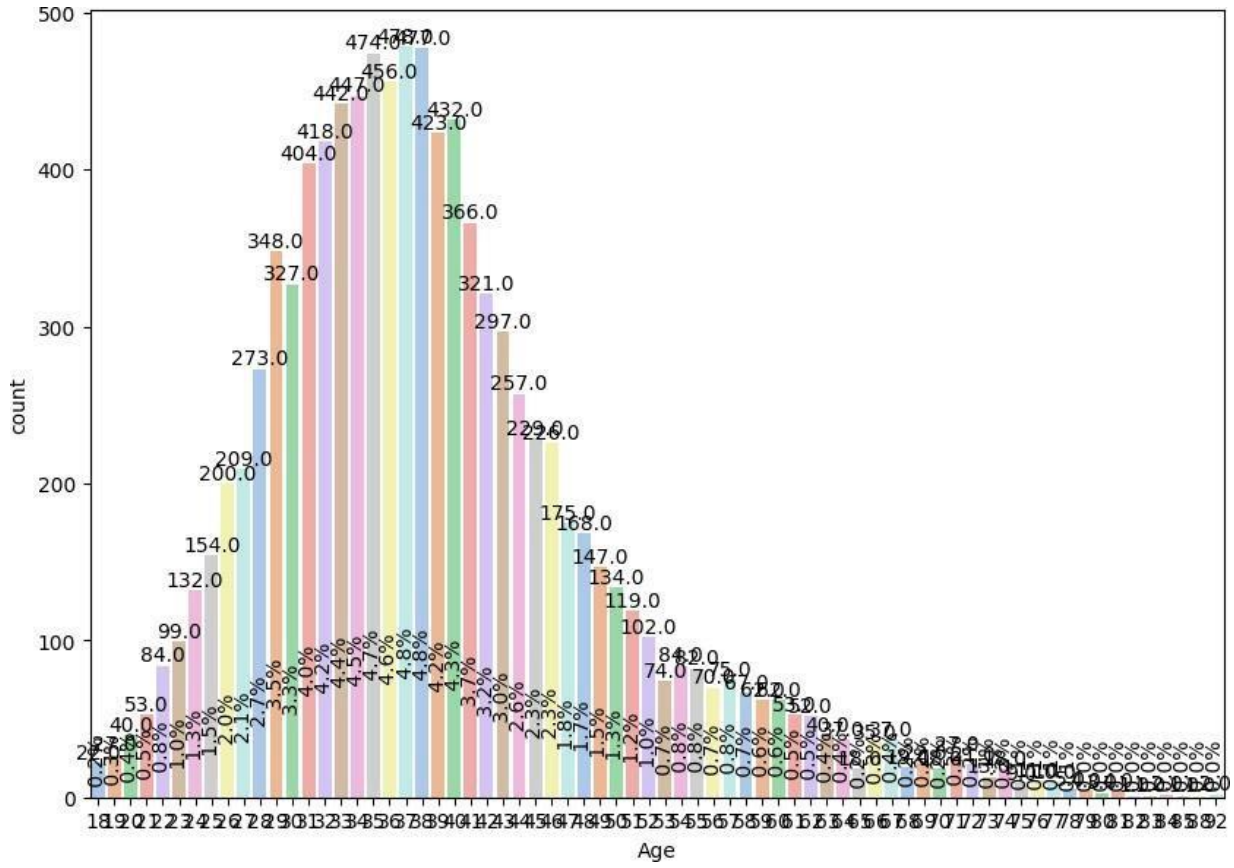
In [ ]:
```
def plot_bivariate(col,hue):
    if(df[col].nunique()>5):
        plt.figure(figsize=(20,10))
        rot=90
    else:
        plt.figure(figsize=(10,7))
        rot=0
    def percentage(ax):
```

```
        heights = [[p.get_height() for p in bars] for bars in ax.containe
        for bars in ax.containers:
            for i, p in enumerate(bars):
                total = sum(group[i] for group in heights) #Sum total of
                percentage = (100 * p.get_height() / total) #Calculate %
                ax.annotate(format(p.get_height()), (p.get_x() + p.get_wi
                    ha = 'center', va = 'bottom', rotation=0)
                if(percentage>25.0):
                    percentage = f'{percentage:.1f}%'
                    ax.annotate( percentage, (p.get_x() + p.get_width()*0


    plot = sns.countplot(x=df[col], hue=df[hue],palette='pastel')
    percentage(plot)
```

In [ ]:
```
def spearman(df,hue):
    feature = []
    correlation = []
    result = []
    for col in df.columns:
        corr, p = stats.spearmanr(df[col], df[hue])
        feature.append(col)
        correlation.append(corr)
        alpha = 0.05
        if p > alpha:
            result.append('No correlation (fail to reject H0)')
        else:
            result.append('Some correlation (reject H0)')
    c = pd.DataFrame({'Feature Name':feature,'correlation coefficient':co
    display(c)
```

In [ ]:
```
plot_univariate('Age')
```

```
/var/folders/sd/gsmj_sw13wj59mxmsl69t0bw0000gn/T/ipykernel_12551/420076724
8.py:10: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be remove
d in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for
the same effect.

  plot = sns.countplot(x = df[col], palette='pastel')
```



In [ ]:
```
plot_bivariate('Age','Exited')
```

In [ ]:
```
spearman(churn_data,'Age')
```

	Feature Name	correlation coefficient	Inference
0	CreditScore	-0.007974	No correlation (fail to reject H0)
1	Geography	0.035351	Some correlation (reject H0)
2	Gender	-0.029785	Some correlation (reject H0)
3	Age	1.000000	Some correlation (reject H0)
4	Tenure	-0.010405	No correlation (fail to reject H0)
5	Balance	0.033304	Some correlation (reject H0)
6	NumOfProducts	-0.058566	Some correlation (reject H0)
7	HasCrCard	-0.015278	No correlation (fail to reject H0)
8	IsActiveMember	0.039839	Some correlation (reject H0)
9	EstimatedSalary	-0.002431	No correlation (fail to reject H0)
10	Exited	0.323968	Some correlation (reject H0)

Label Encoding of Categorical Variables

Label Encoding means converting categorical features into numerical values. So that they can be fitted by machine learning models which only take numerical data.

Example: Suppose we have a column Height in some dataset that has elements as Tall, Medium, and short. To convert this categorical column into a numerical column we will apply label encoding to this column. After applying label encoding, the Height column is converted into a numerical column having elements 0,1, and 2 where 0 is the label for tall, 1 is the label for medium, and 2 is the label for short height.

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
churn_data[['Geography', 'Gender']] = churn_data[['Geography', 'Gender']]
```

In [ ]:
```
churn_data.head()
```

Out[ ]:

RowNumber	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfPro
1	619	0	0	42	2	0.00	
2	608	2	0	41	1	83807.86	
3	502	0	0	42	8	159660.80	
4	699	0	0	39	1	0.00	
5	850	2	0	43	2	125510.82	

Seperating Label from Data

In [ ]:
```
y = churn_data.Exited
X = churn_data.drop(['Exited'],axis=1)
```

In [ ]:
```
X.columns
```

Out[ ]:
```
Index(['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance',
       'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'EstimatedSalar
y'],
      dtype='object')
```

In [ ]:
```
y
```

Splitting the Data into Training and

Testing

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3
```

In [ ]:
```
print("Shape of the X_train", X_train.shape)
print("Shape of the X_test", X_test.shape)
print("Shape of the y_train", y_train.shape)
print("Shape of the y_test", y_test.shape)
```

```
Shape of the X_train (7000, 10)
Shape of the X_test (3000, 10)
Shape of the y_train (7000,)
Shape of the y_test (3000,)
```

Feature Scaling:

The result of standardization (or Z-Score normalization) is that the features will be re scaled so that they'll have the properties of a standard normal distribution with:

$$\mu = 0$$

And

$$\sigma = 1$$

Where μ is the mean(average) and σ is the standard deviation from the mean; standard scores (also called Z scores) of the sampels are calculated as follows:

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

About Min-Max Scaling

An alternative approach to Z-Score normalization (or called standardization) is the so-called Min-Max Scaling (often also simply called Normalization - a common cause for ambiguities)

In this approach, the data is scaled to a fixed range - usually `[0, 1]` . The cost of having this bounded range - in contrast to standrdization - is that we will end up with smaaller standard deviations, which can suppress the effect of outliers.

Note:

If the dataset have lot's of outliers, and the algorithms are sensitive to outliers,

please use `Min-Max Scaler`

A `Min-Max Scaling` is typically done via the foloowing equation:

$$X_{norm} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

X_i is the i^{th} sample of dataset.

Z-Score Standardization or Min-Max Scaling

"Standardization or Min-Max scaling"? - There is no obvious answer to this question: it really depends on the application.

However this doesn't mean that `Min-Max Scaling` is not useful at all, A popular application is `image processing` , where pixel intensities have to be normalized to fit withint a certain range (i.e., `[0, 255]` for the RGB colour range). Also, typical *Neural Network* Algorithm require data that on a `0 - 1` scale.

Need for Normalization

For example, consider a data set containing two features, age(x1), and income(x2). Where age ranges from 0-100, while income ranges from 0-20,000 and higher. Income is about 1,000 times larger than age and ranges from 20,000-500,000. So, these two features are in very different ranges. When we do further analysis, like multivariate linear regression, for example, the attributed income will intrinsically influence the result more due to its larger value. But this doesn't necessarily mean it is more important as a predictor.

In [ ]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Building the ANN Model

In [ ]:
```
# sequential model to initialise our ann and dense module to build the la
from keras.models import Sequential
from keras.layers import Dense
```

In [ ]:
```
classifier = Sequential()
# Adding the input layer and the first hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activatio

# Adding the second hidden layer
classifier.add(Dense(units = 16, kernel_initializer = 'uniform', activati

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activatio
```

Compiling and Fitting the Model

In [ ]:
```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metr

# Fitting the ANN to the Training set
classifier.fit(X_train, y_train, batch_size = 10, epochs = 100, verbose =
```

Testing the Model

In [ ]:
```
score, acc = classifier.evaluate(X_train, y_train,
                            batch_size=10)
print('Train score:', score)
print('Train accuracy:', acc)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.5)

print('*'*20)
score, acc = classifier.evaluate(X_test, y_test,
                            batch_size=10)
print('Test score:', score)
print('Test accuracy:', acc)
```

```
700/700 [==============================] - 0s 278us/step - loss: 0.3287 -
accuracy: 0.8637
Train score: 0.328686386346817
Train accuracy: 0.8637142777442932
94/94 [==============================] - 0s 277us/step
********************
300/300 [==============================] - 0s 300us/step - loss: 0.3470 -
accuracy: 0.8593
Test score: 0.34699997305870056
Test accuracy: 0.859333336353302
```

Confusion Matrix

* Accuracy

number of examples correctly predicted / total number of examples

$$ACC \quad (TP + TN)/(TP + FP + FN + TN)$$

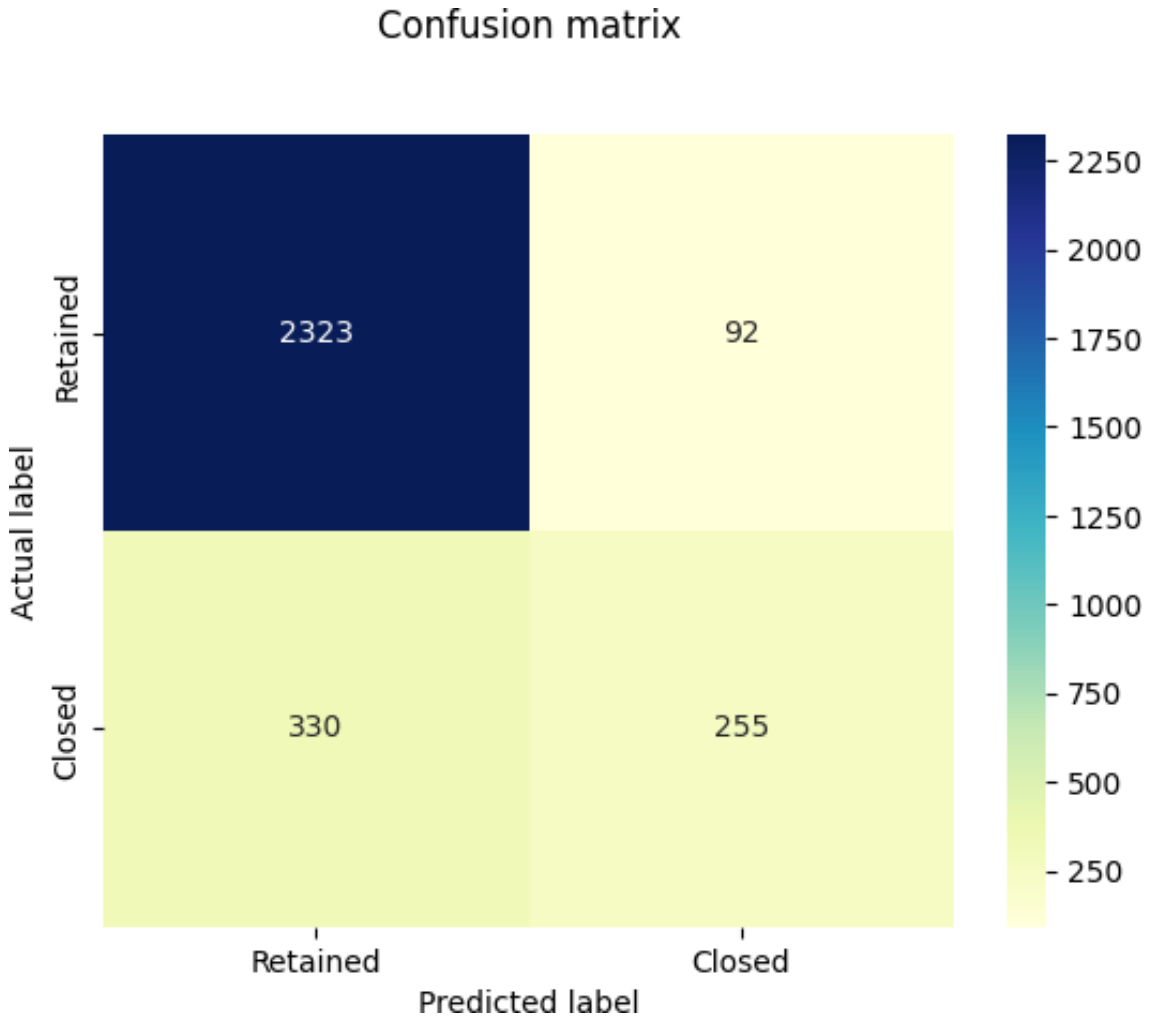
In [ ]:
```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
target_names = ['Retained', 'Closed']
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[2323   92]
 [ 330  255]]
```

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:
```
p = sns.heatmap(pd.DataFrame(cm), annot=True, xticklabels=target_names, y
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Out[ ]:
```
Text(0.5, 23.52222222222222, 'Predicted label')
```



Classification Report

* True Positive Rate

number of samples actually and predicted as `Positive` / total number of samples actually `Positive`

Also called Sensitivity or Recall.

$$TPR = TP/P = TP/(TP + FN)$$

* Positive Predictive Value

number of samples actually and predicted as `Positive` / total number of samples predicted as `Positive`

Also called Precision.

$$PPV = TP/(TP + FP)$$

* F1 score

Harmonic Mean of Precision and Recall.

$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

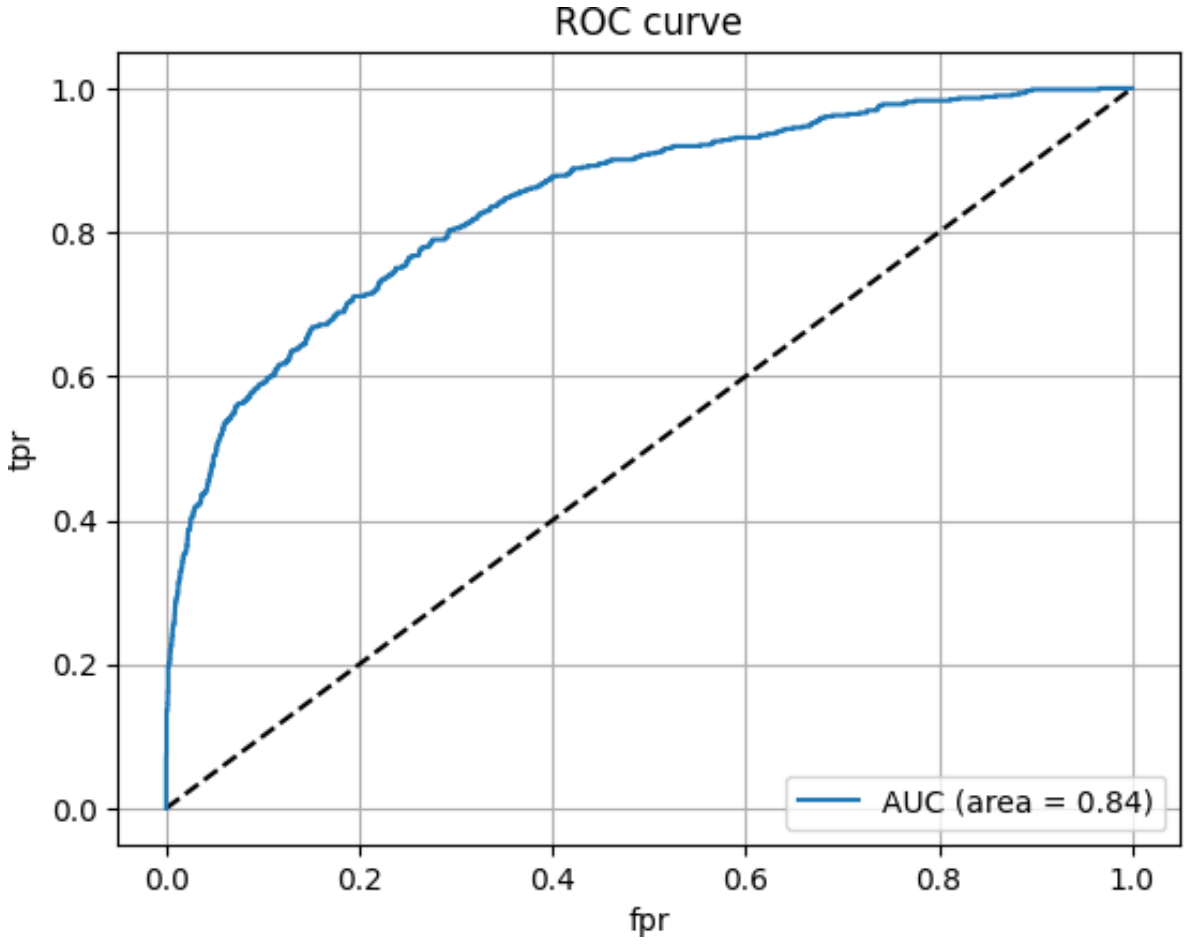
In [ ]:
```
#import classification_report
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred, target_names=target_names))
```

```
              precision    recall  f1-score   support

    Retained       0.88      0.96      0.92      2415
      Closed       0.73      0.44      0.55       585

    accuracy                           0.86      3000
   macro avg       0.81      0.70      0.73      3000
weighted avg       0.85      0.86      0.84      3000
```

ROC curve

In [ ]:
```
from sklearn.metrics import roc_curve, auc
y_pred_proba = classifier.predict(X_test)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='AUC (area = %0.2f)' % roc_auc)
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.grid()
plt.legend(loc="lower right")
plt.title('ROC curve')
plt.show()
```

```
94/94 [==============================] - 0s 314us/step
```



In [ ]:
```
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred_proba)
```

Out[ ]: 0.8436201801419193

Experiment : 3

AIM: Build an Artificial Neural Network to implement Multi-Class Classification task using the Back-propagation algorithm and test the same using appropriate data sets.

Importing Necessary Libraries

In [ ]:
```
import numpy as np
import pandas as pd
```

Loading the Combined Cycle Power Plant Dataset

The dataset contains 9568 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011), when the power plant was set to work with full load. Features consist of hourly average ambient variables Temperature (AT), Ambient Pressure (AP), Relative Humidity (RH) and Exhaust Vacuum (V) to predict the net hourly electrical energy output (PE) of the plant.

- For more information visit below link: Link: https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant

In [ ]:
```
Powerplant_data = pd.read_excel('Folds5x2_pp.xlsx')
Powerplant_data.head(5)
```

Out[ ]:

	AT	V	AP	RH	PE
0	14.96	41.76	1024.07	73.17	463.26
1	25.18	62.96	1020.04	59.08	444.37
2	5.11	39.40	1012.16	92.14	488.56
3	20.86	57.32	1010.24	76.64	446.48
4	10.82	37.50	1009.23	96.62	473.90

Accessing the Column Names in the

Dataset

In [ ]:
```
Powerplant_data.columns
```

Out[ ]:
```
Index(['AT', 'V', 'AP', 'RH', 'PE'], dtype='object')
```

Finding the Shape of the Dataset

In [ ]:
```
Powerplant_data.shape
```

Out[ ]:
```
(9568, 5)
```

In [ ]:
```
Powerplant_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9568 entries, 0 to 9567
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AT      9568 non-null   float64
 1   V       9568 non-null   float64
 2   AP      9568 non-null   float64
 3   RH      9568 non-null   float64
 4   PE      9568 non-null   float64
dtypes: float64(5)
memory usage: 373.9 KB
```

Checking Missing Values

In [ ]:
```
Powerplant_data.isna().sum()
```

Out[ ]:
```
AT    0
V     0
AP    0
RH    0
PE    0
dtype: int64
```

Number of Unique Items in Each Column

In [ ]:
```
Powerplant_data.nunique()
```

Out[ ]:
```
AT    2773
V      634
AP    2517
RH    4546
PE    4836
dtype: int64
```

Seperating Label from Data

In [ ]:
```
X = Powerplant_data.iloc[:, :-1].values
y = Powerplant_data.iloc[:, -1].values
```

or

In [ ]:
```
# y o Powerplant_data['PE']
# X o Powerplant_data.drop(['PE'],axis=1)
```

Splitting the Data into Training and Testing

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2
X_train, X_val, y_train, y_val = train_test_split(X_train,y_train, test_s
```

In [ ]:
```
print("Shape of the X_train", X_train.shape)
print("Shape of the X_test", X_test.shape)
print("Shape of the X_val", X_val.shape)
print("Shape of the y_train", y_train.shape)
print("Shape of the y_test", y_test.shape)
print("Shape of the y_val", y_val.shape)
```

```
Shape of the X_train (6123, 4)
Shape of the X_test (1914, 4)
Shape of the X_val (1531, 4)
Shape of the y_train (6123,)
Shape of the y_test (1914,)
Shape of the y_val (1531,)
```

Building the ANN Model

In [ ]:
```
# sequential model to initialise our ann and dense module to build the la
from keras.models import Sequential
from keras.layers import Dense
```

In [ ]:
```
classifier = Sequential()
```

```
# Adding the input layer and the first hidden layer
classifier.add(Dense(units = 8, kernel_initializer = 'uniform', activatio

# Adding the second hidden layer
classifier.add(Dense(units = 16, kernel_initializer = 'uniform', activati

# Adding the third hidden layer
classifier.add(Dense(units = 32, kernel_initializer = 'uniform', activati

# Adding the output layer
classifier.add(Dense(units = 1, kernel_initializer = 'uniform'))
```

Compiling and Fitting the Model

In [ ]:
```
classifier.compile(optimizer = 'adam', loss = 'mean_squared_error', metri

# Fitting the ANN to the Training set
model = classifier.fit(X_train, y_train, batch_size = 32, epochs = 200,va
                       shuffle=True)
```

Testing the Model

In [ ]:
```
y_pred = classifier.predict(X_test)
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y
```

```
60/60 [==============================] - 0s 293us/step
[[463.95 471.81]
 [443.18 444.61]
 [437.15 437.03]
 ...
 [440.91 442.77]
 [441.47 445.33]
 [432.53 432.94]]
```

Metric values

MAE (Mean Absolute Error) :-

$$MAE = \left(\frac{1}{n}\right)\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MSE (Mean Square Error) :-

$$MSE = \left(\frac{1}{n}\right)\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y = actual value in the data set ; $\hat{y}$ = value computed by solving the regression equation

RMSE (Root Mean Square Error) :-

$$RMSE = \sqrt{\left(\frac{1}{n}\right)\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:
```
import sklearn.metrics
from math import sqrt
mae_no = sklearn.metrics.mean_absolute_error(y_test,classifier.predict(X_
mse_no = sklearn.metrics.mean_squared_error(y_test,classifier.predict(X_t
rms = sqrt(sklearn.metrics.mean_squared_error(y_test,classifier.predict(X
```

```
60/60 [==============================] - 0s 678us/step
60/60 [==============================] - 0s 263us/step
60/60 [==============================] - 0s 269us/step
```

In [ ]:
```
print('Mean Absolute Error      :',mae_no)
print('Mean Square Error        :',mse_no)
print('Root Mean Square Error:', rms)
```

```
Mean Absolute Error      : 4.156925082630249
Mean Square Error        : 28.931698846583828
Root Mean Square Error: 5.37881946588504
```

RESULT: We have succefully build an Artificial Neural Network to implement Multi-Class Classification task using the Back-propagation algorithm and test the same using appropriate data sets.

Experiment : 4

AIM: Implement an image classification task using pre-trained models like AlexNet, VGGNet, InceptionNet and ResNet and compare the results.

Importing Necessary libraries

- From keras library we are going to use image preprocessing task, to normalize the image pixel values in between 0 to 1.
- Model is imported to load variuos Neural Netwrok models such as Sequential.
- We are going to use transfer learning technique.

In [ ]:
```
import keras
import numpy as np
from keras  import  Input
from keras import models
from keras import layers
from keras import optimizers
from keras.models  import  Model
from keras  import  applications
from keras  import backend  as  k
import matplotlib.pyplot as plt
from keras.optimizers import SGD, Adam
from keras.layers import  Dense, Flatten, Conv2D, MaxPooling2D
from keras.preprocessing import image
from keras.models import Sequential, Model
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, Tenso
```

Loading the Training and Testing Data and Defining the Basic Parameters

- We are resizing the input image to 128 * 128
- In the dataset : Training Set : 70% Validation Set : 20% Test Set : 10%

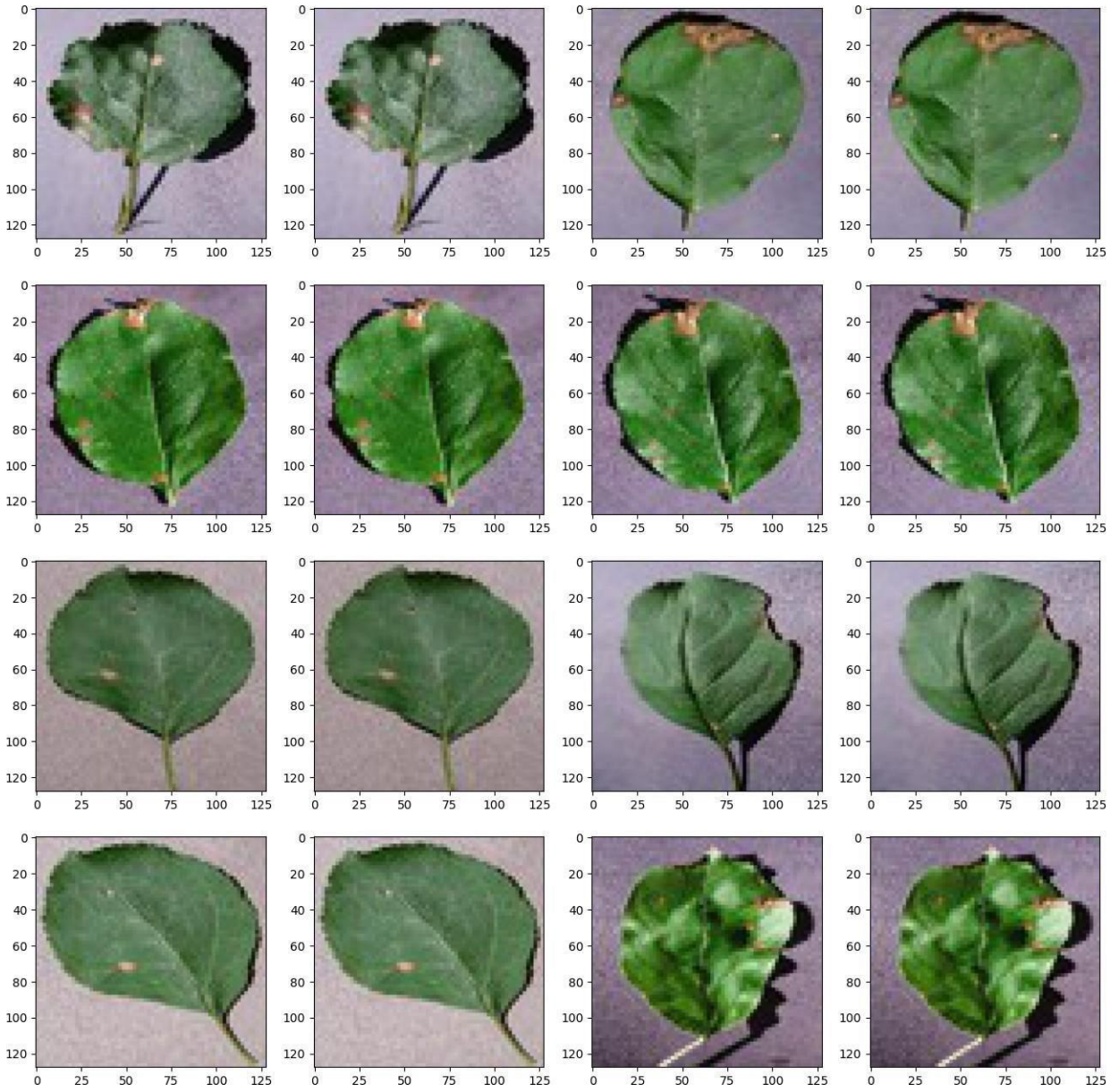
In [ ]:
```
# Normalize training and validation data in the range of 0 to 1
train_datagen = ImageDataGenerator(rescale=1./255) # vertical_flip=True,
                                                   # horizontal_flip=True
```

```
                                                    # height_shift_range=0
                                                    # width_shift_range=0.
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Read the training sample and set the batch size
train_generator = train_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(128, 128),
        batch_size=16,
        class_mode='categorical')

# Read Validation data from directory and define target size with batch s
validation_generator = validation_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(128, 128),
        batch_size=16,
        class_mode='categorical',
        shuffle=False)

test_generator = test_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(128, 128),
        batch_size=1,
        class_mode='categorical',
        shuffle=False)
```

```
Found 3002 images belonging to 4 classes.
Found 635 images belonging to 4 classes.
Found 546 images belonging to 4 classes.
```

Visualization of Few Images

In [ ]:
```
plt.figure(figsize=(16, 16))
for i in range(1, 17):
  plt.subplot(4, 4, i)
  img, label = test_generator.next()
  # print(img.shape)
  # print(label)
  plt.imshow(img[0])
plt.show()
```



In [ ]:
```
img, label = test_generator.next()
img[0].shape
```

Out[ ]: (128, 128, 3)

What is ImageNet?

ImageNet is formally a project aimed at (manually) labeling and categorizing images into almost 22,000 separate object categories for the purpose of computer vision research.

However, when we hear the term “ImageNet” in the context of deep learning and Convolutional Neural Networks, we are likely referring to the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC for short.

The goal of this image classification challenge is to train a model that can correctly classify an input image into 1,000 separate object categories.

Models are trained on ~1.2 million training images with another 50,000 images for validation and 100,000 images for testing.

Exploring Keras Applications for Transfer Learning

VGG16

In [ ]:
```
from tensorflow.keras.applications.vgg16 import VGG16

## Loading VGG16 model
base_model = VGG16(weights="imagenet", include_top=False, input_shape= (1
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

Adding top layers according to number of classes in our data

In [ ]:
```
flatten_layer = layers.GlobalAveragePooling2D()
# dense_layer_1 = layers.Dense(63, activation='relu')
# dense_layer_2 = layers.Dense(32, activation='relu')
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    prediction_layer
])

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 4, 4, 512)         14714688

 global_average_pooling2d_2  (None, 512)               0
 (GlobalAveragePooling2D)

 dense_2 (Dense)             (None, 4)                 2052

=================================================================
Total params: 14,716,740
Trainable params: 2,052
Non-trainable params: 14,714,688
_________________________________________________________________
```

Training

In [ ]:
```
# sgd = SGD(lr=0.001,decay=1e-6, momentum=0.9, nesterov=True)
# We are going to use accuracy metrics and cross entropy loss as performa
model.compile(optimizer = Adam(learning_rate = 0.001), loss='categorical_
# Train the model
history = model.fit(train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=validation_generator.samples/validation_generator.
      verbose=1)
```

Saving the model

In [ ]:
```
model.save("VGG16_plant_deseas.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

Loading the model

In [ ]:
```
model = models.load_model('VGG16_plant_deseas.h5')
print("Model is loaded")
```

```
Model is loaded
```

Saving the Weights

In [ ]:
```
model.save_weights('cnn_classification.h5')
```

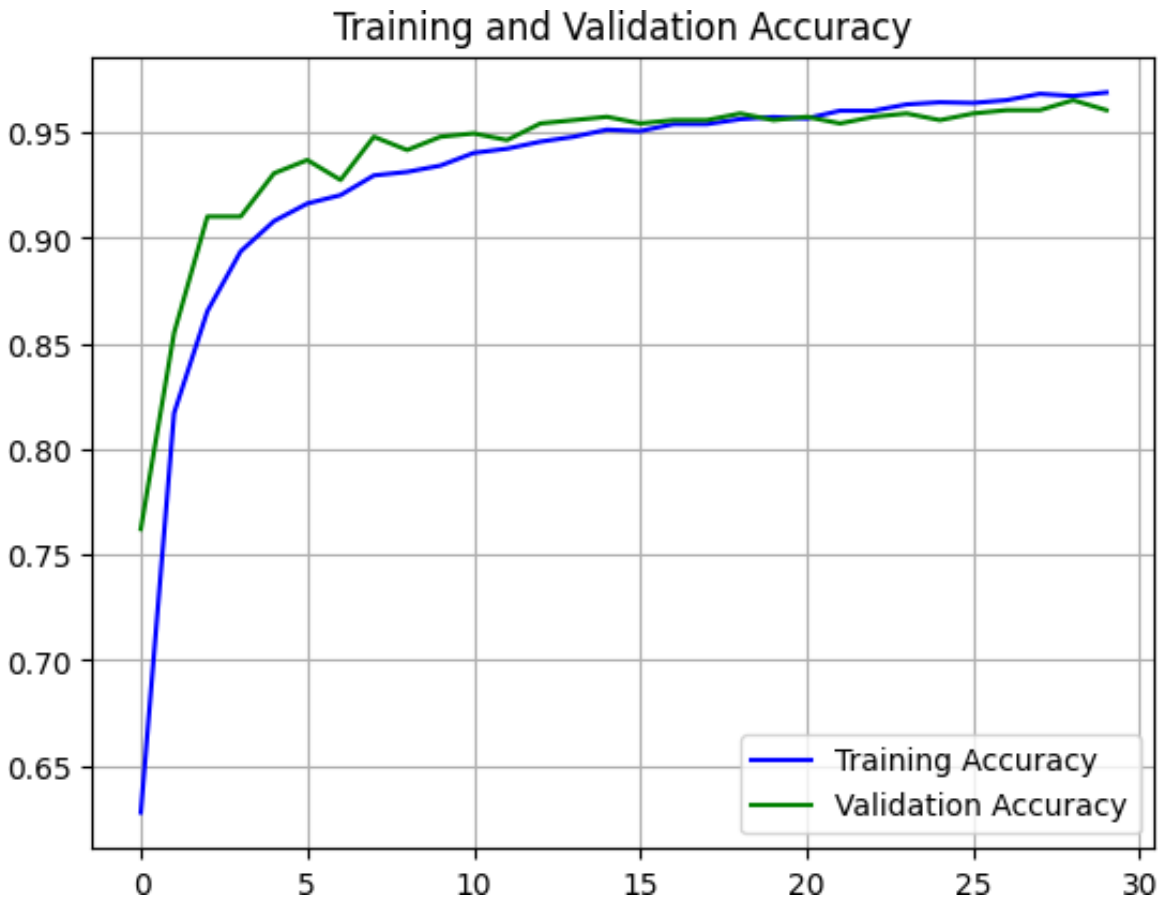
Loading the weights

In [ ]:
```
model.load_weights('cnn_classification.h5')
```

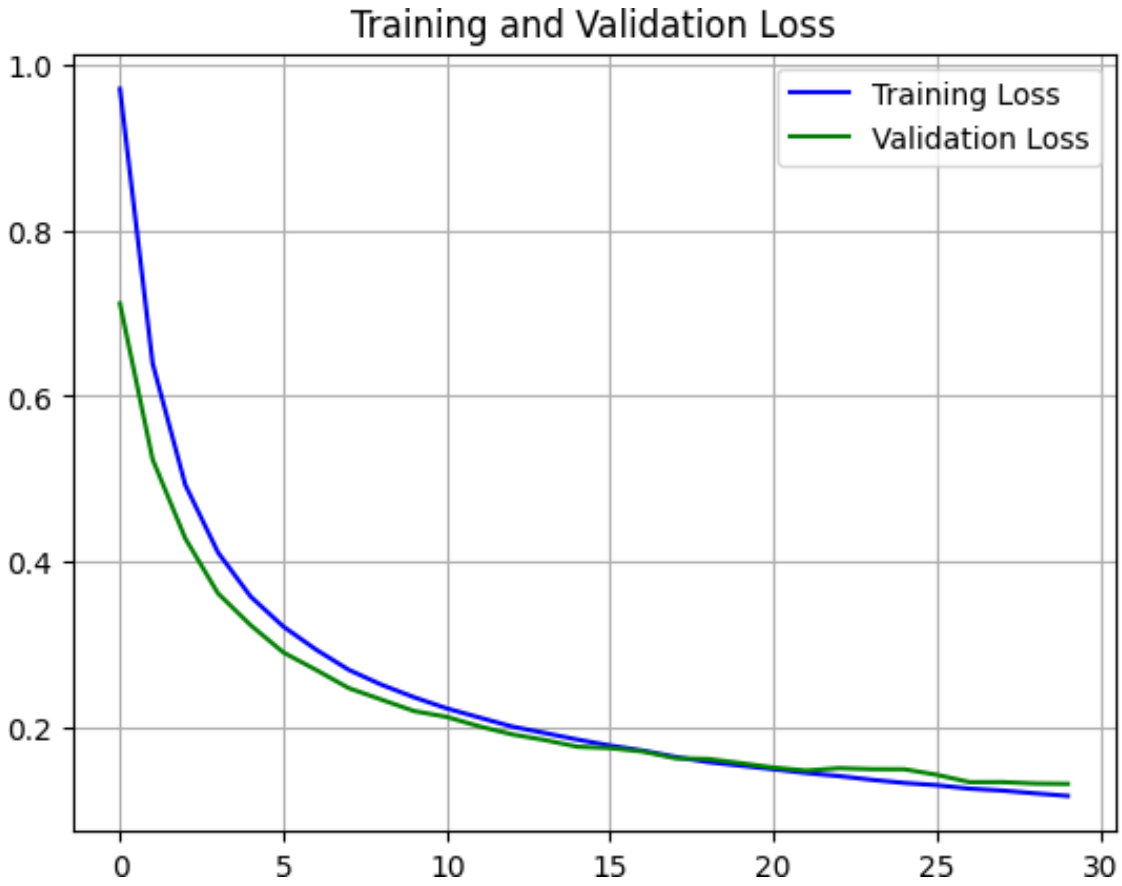
Visualization of training over epoch

In [ ]:
```
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
```

In [ ]:
```
epochs = range(len(train_acc))
plt.plot(epochs, train_acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.legend()
plt.figure()
plt.show()

plt.plot(epochs, train_loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'g', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.grid()
plt.legend()
plt.show()
```




```
<Figure size 640x480 with 0 Axes>
```



Performance measure

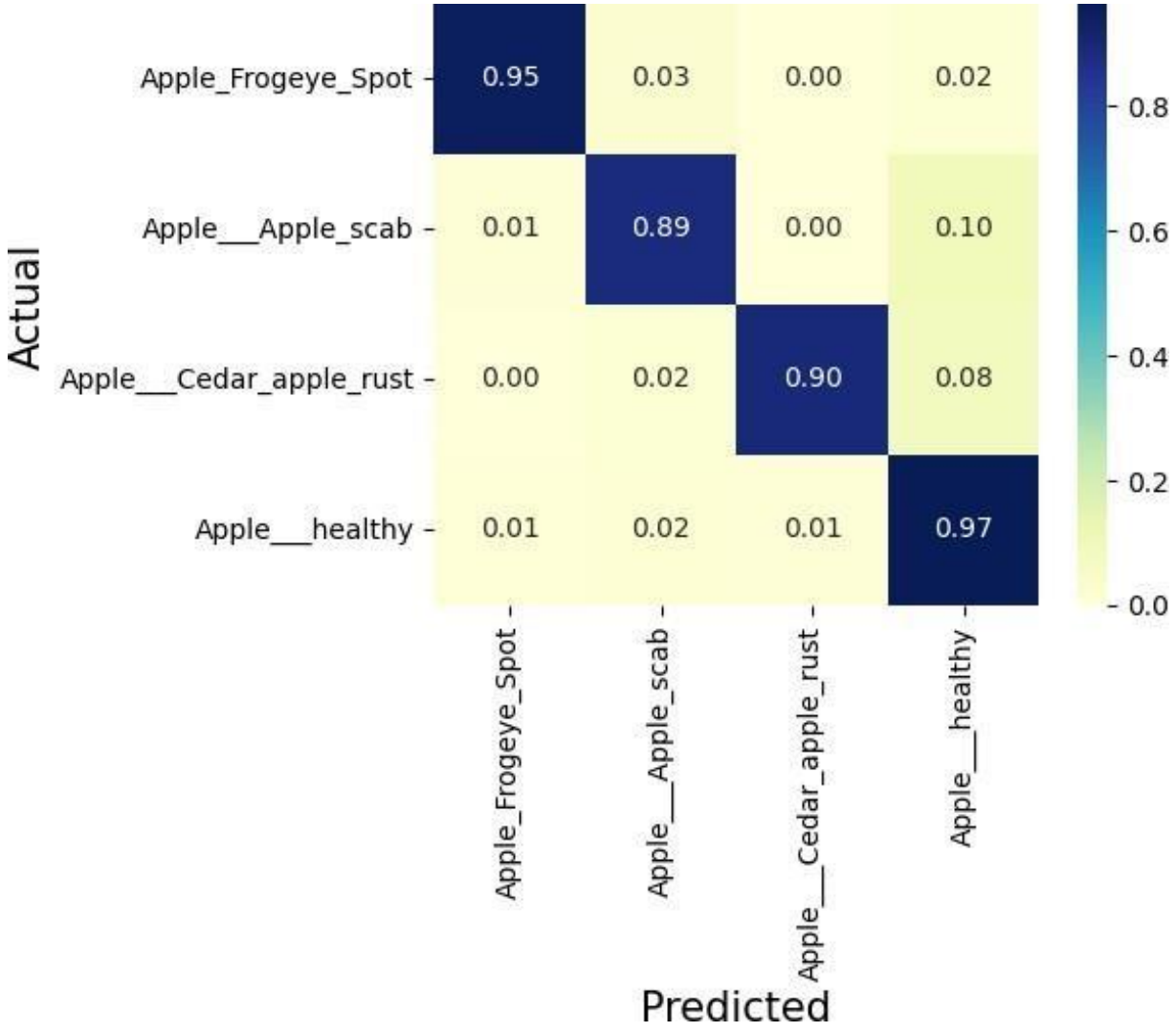
In [ ]:
```
# Get the filenames from the generator
fnames = test_generator.filenames

# Get the ground truth from generator
ground_truth = test_generator.classes

# Get the label to class mapping from the generator
label2index = test_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(test_generator, steps=test_generato
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),test_generator.samples))
```

```
<ipython-input-20-1bdf1cd647ce>:14: UserWarning: `Model.predict_generator`
is deprecated and will be removed in a future version. Please use `Model.p
redict`, which supports generators.
  predictions = model.predict_generator(test_generator, steps=test_generat
or.samples/test_generator.batch_size,verbose=1)
546/546 [==============================] - 3s 5ms/step
No of errors = 34/546
```

In [ ]:
```
accuracy = ((test_generator.samples-len(errors))/test_generator.samples)
accuracy
```

Out[ ]: 93.77289377289377

In [ ]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
cm = confusion_matrix(y_true=ground_truth, y_pred=predicted_classes)
cm = np.array(cm)
# Normalise
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=label2index, yticklab
plt.ylabel('Actual', fontsize=15)
plt.xlabel('Predicted', fontsize=15)
plt.show(block=False)
```



In [ ]:
```
from sklearn.metrics import classification_report
print(classification_report(ground_truth, predicted_classes, target_names
```

	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.96	0.95	0.96	103
Apple___Apple_scab	0.94	0.89	0.91	134
Apple___Cedar_apple_rust	0.94	0.90	0.92	49
Apple___healthy	0.93	0.97	0.95	260
accuracy			0.94	546
macro avg	0.94	0.93	0.93	546
weighted avg	0.94	0.94	0.94	546

In [ ]:

InceptionNet

In [ ]:
```
from keras import applications
```

```
## Loading InceptionV3 model
base_model = applications.InceptionV3(weights="imagenet", include_top=Fal
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

In [ ]:
```
flatten_layer = layers.GlobalAveragePooling2D()
# dense_layer_1 = layers.Dense(63, activation='relu')
# dense_layer_2 = layers.Dense(32, activation='relu')
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    # dense_layer_1,
    # dense_layer_2,
    prediction_layer
])

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inception_v3 (Functional)   (None, 2, 2, 2048)        21802784

 global_average_pooling2d_3  (None, 2048)              0
 (GlobalAveragePooling2D)

 dense_3 (Dense)             (None, 4)                 8196

=================================================================
Total params: 21,810,980
Trainable params: 8,196
Non-trainable params: 21,802,784
_________________________________________________________________
```

In [ ]:
```
model.compile(optimizer = Adam(learning_rate = 0.001), loss='categorical_
# Train the model
history = model.fit(train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=validation_generator.samples/validation_generator.
      verbose=1)
```

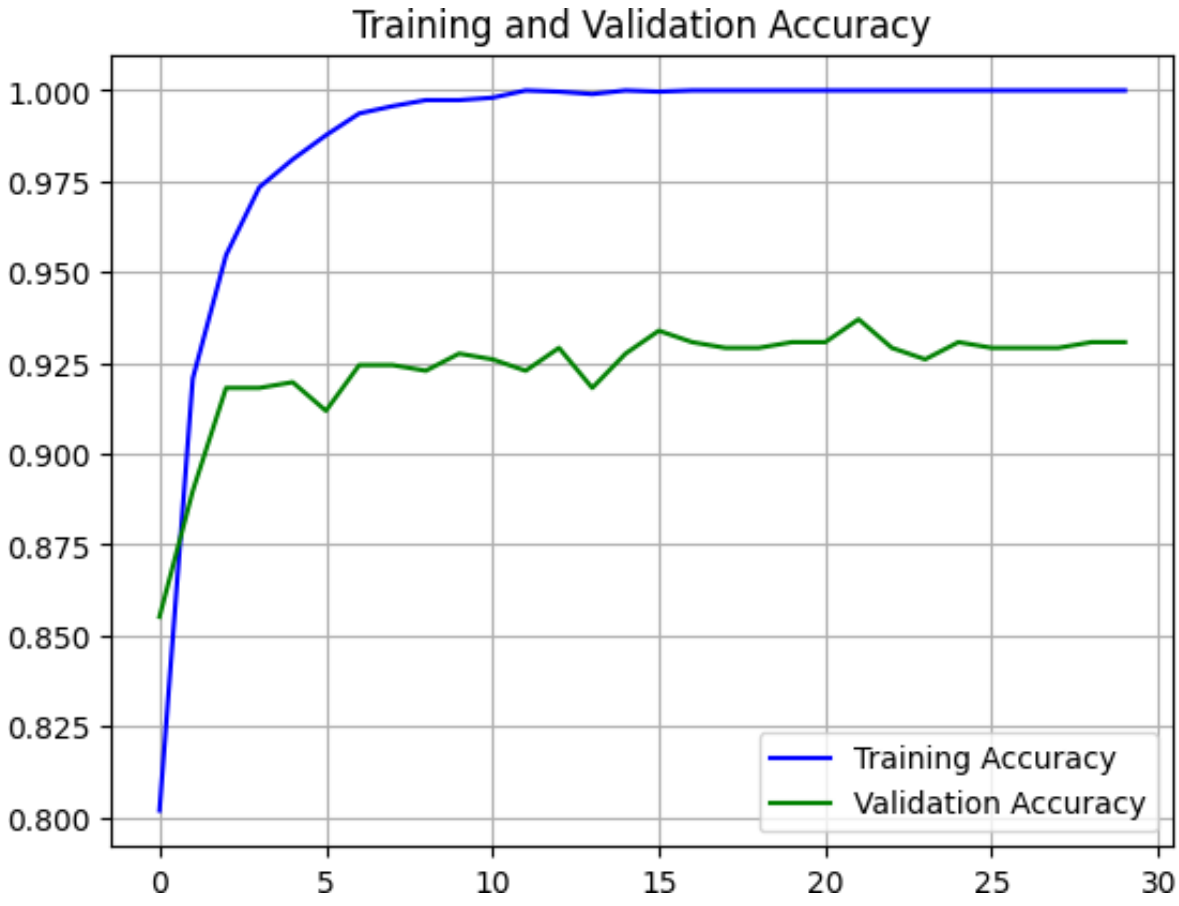
In [ ]:
```
model.save("InceptionNet_plant_deseas.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

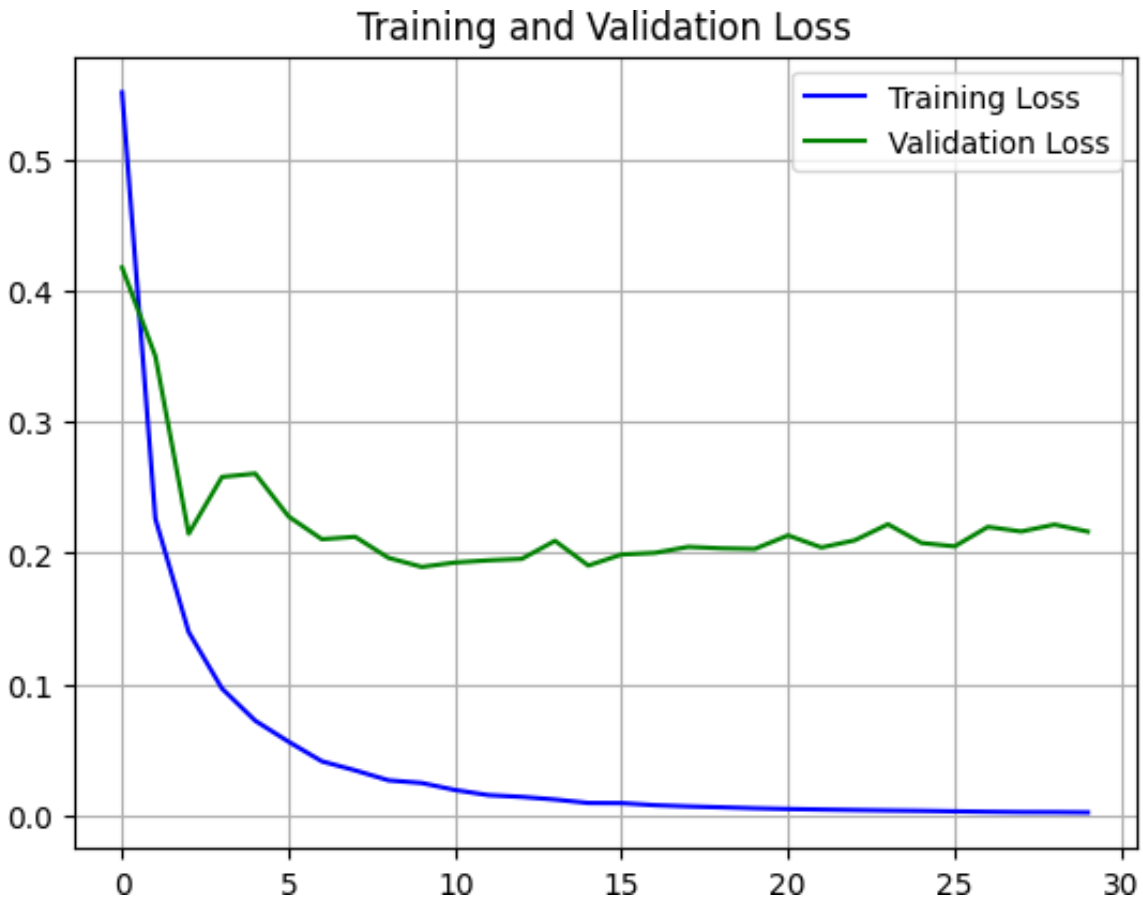
In [ ]:
```
train_acc = history.history['acc']
```

```
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
```

In [ ]:
```
epochs = range(len(train_acc))
plt.plot(epochs, train_acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.legend()
plt.figure()
plt.show()

plt.plot(epochs, train_loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'g', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.grid()
plt.legend()
plt.show()
```




```
<Figure size 640x480 with 0 Axes>
```



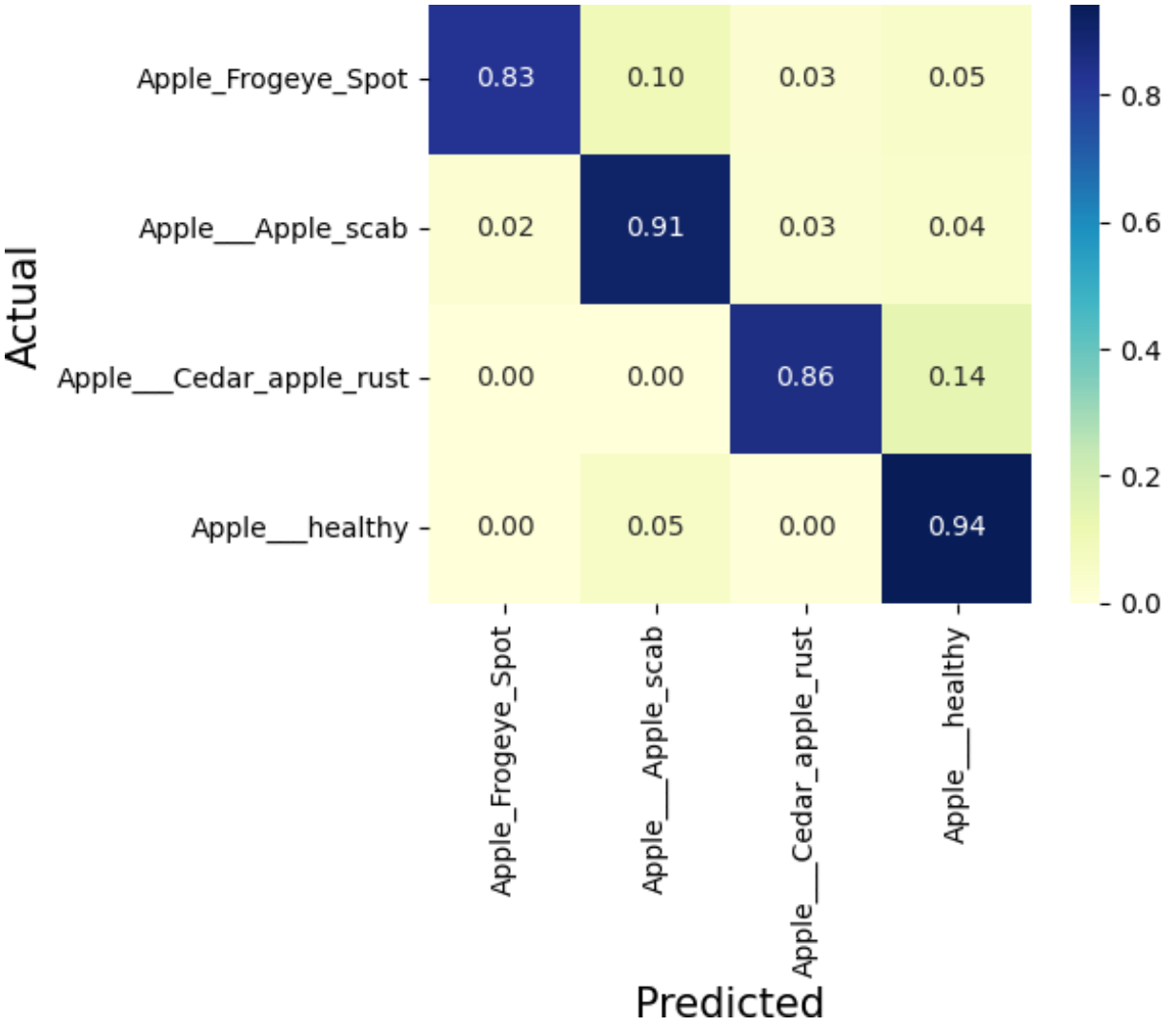
In [ ]:
```
# Get the filenames from the generator
fnames = test_generator.filenames

# Get the ground truth from generator
ground_truth = test_generator.classes

# Get the label to class mapping from the generator
label2index = test_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(test_generator, steps=test_generato
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),test_generator.samples))
```

```
<ipython-input-30-1bdf1cd647ce>:14: UserWarning: `Model.predict_generator`
is deprecated and will be removed in a future version. Please use `Model.p
redict`, which supports generators.
  predictions = model.predict_generator(test_generator, steps=test_generat
or.samples/test_generator.batch_size,verbose=1)
546/546 [==============================] - 8s 11ms/step
No of errors = 52/546
```

In [ ]:
```
accuracy = ((test_generator.samples-len(errors))/test_generator.samples)
accuracy
```

Out[ ]: 90.47619047619048

In [ ]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
cm = confusion_matrix(y_true=ground_truth, y_pred=predicted_classes)
cm = np.array(cm)
# Normalise
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=label2index, yticklab
plt.ylabel('Actual', fontsize=15)
plt.xlabel('Predicted', fontsize=15)
plt.show(block=False)
```



In [ ]:
```
from sklearn.metrics import classification_report
print(classification_report(ground_truth, predicted_classes, target_names
```

	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.97	0.83	0.89	103
Apple___Apple_scab	0.84	0.91	0.87	134
Apple___Cedar_apple_rust	0.84	0.86	0.85	49
Apple___healthy	0.94	0.94	0.94	260
accuracy			0.90	546
macro avg	0.89	0.88	0.89	546
weighted avg	0.91	0.90	0.90	546

In [ ]:

ResNet

In [ ]:
```
from keras import applications

## Loading VGG16 model
base_model = applications.ResNet50(weights="imagenet", include_top=False,
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

In [ ]:
```
flatten_layer = layers.GlobalAveragePooling2D()
# dense_layer_1 = layers.Dense(63, activation='relu')
# dense_layer_2 = layers.Dense(32, activation='relu')
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    # dense_layer_1,
    # dense_layer_2,
    prediction_layer
])

model.summary()
```

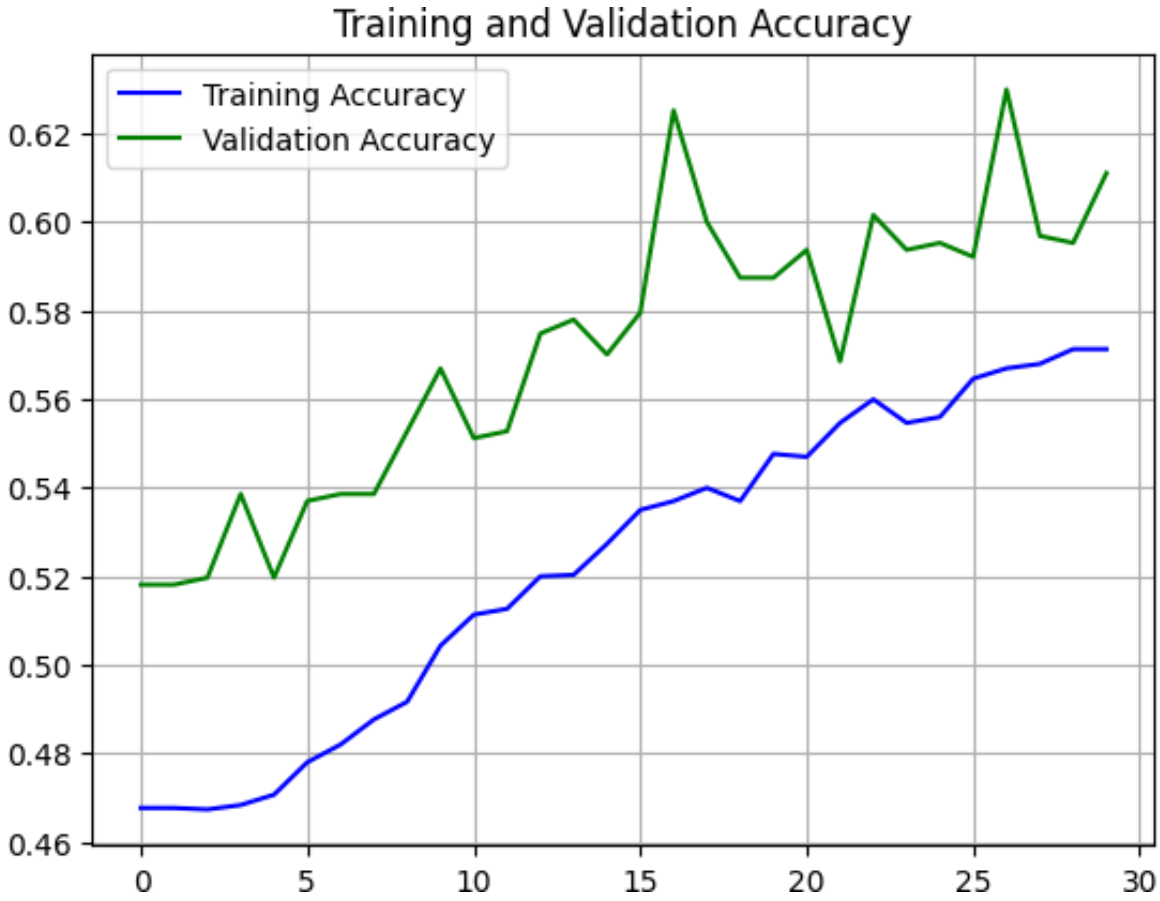
In [ ]:
```
model.compile(optimizer = Adam(learning_rate = 0.001), loss='categorical_
# Train the model
history = model.fit(train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=validation_generator.samples/validation_generator.
      verbose=1)
```

In [ ]:
```
model.save("ResNet_plant_deseas.h5")
print("Saved model to disk")
```

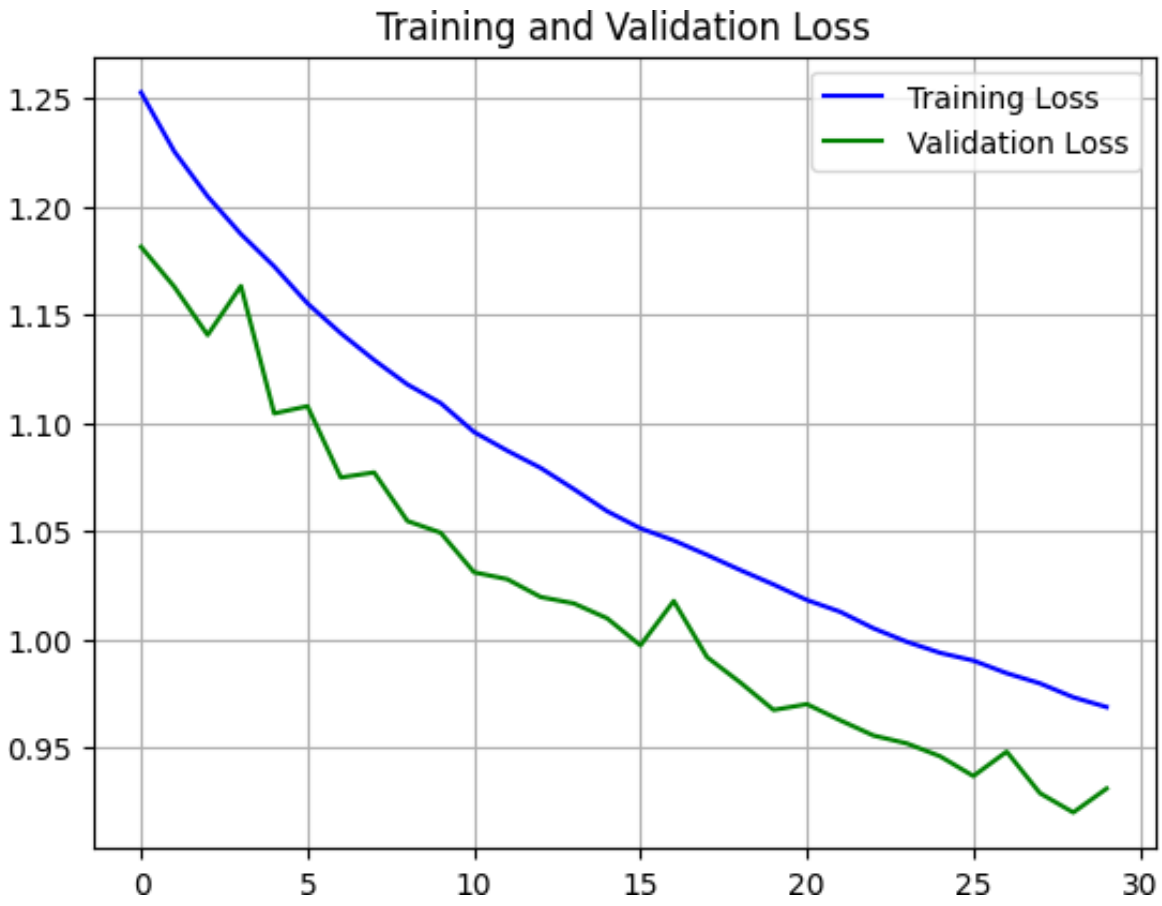
```
Saved model to disk
```

In [ ]:
```
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
```

In [ ]:
```
epochs = range(len(train_acc))
plt.plot(epochs, train_acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.legend()
plt.figure()
plt.show()

plt.plot(epochs, train_loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'g', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.grid()
plt.legend()
plt.show()
```




```
<Figure size 640x480 with 0 Axes>
```



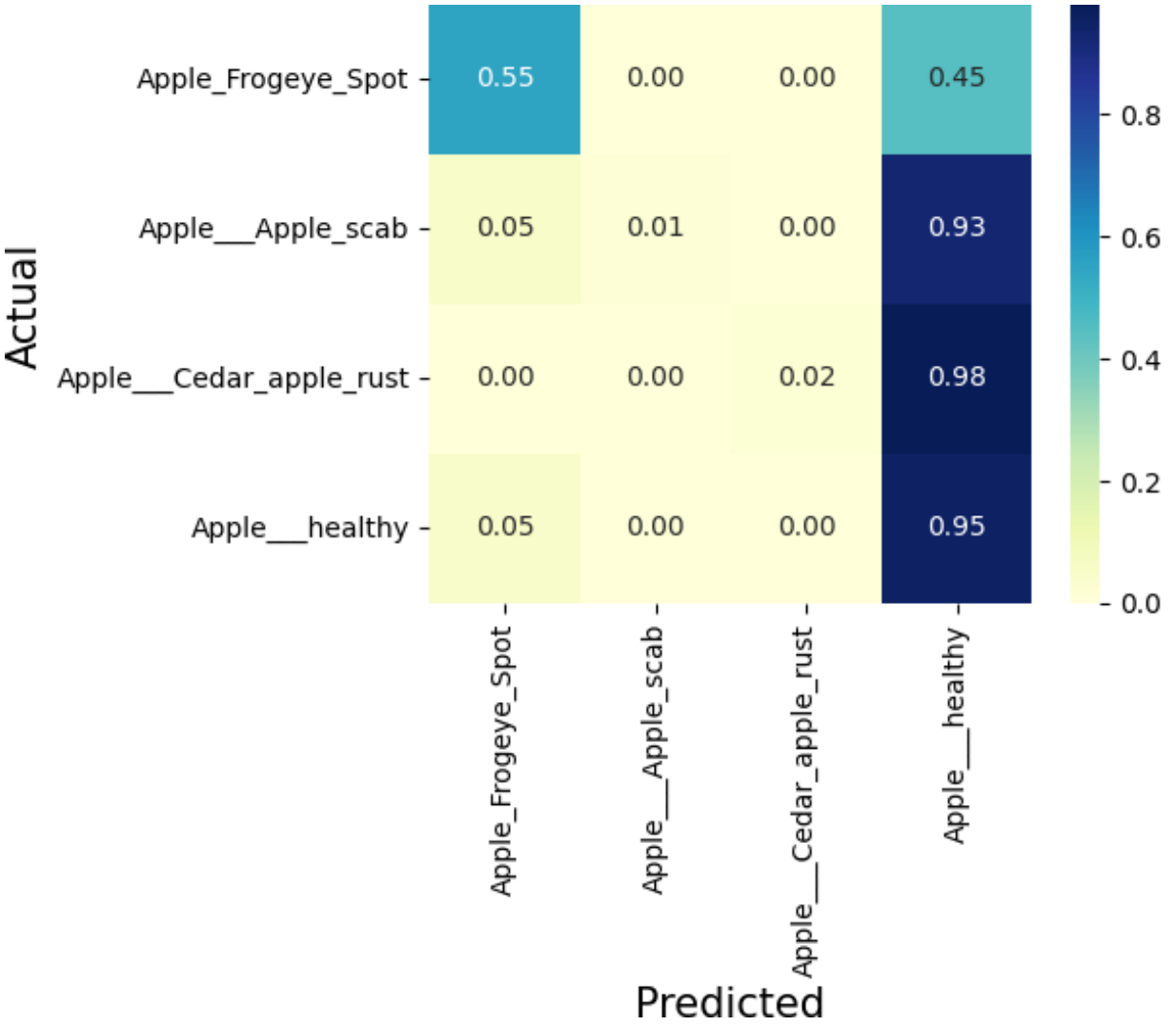
In [ ]:
```
# Get the filenames from the generator
fnames = test_generator.filenames

# Get the ground truth from generator
ground_truth = test_generator.classes

# Get the label to class mapping from the generator
label2index = test_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(test_generator, steps=test_generato
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),test_generator.samples))
```

```
<ipython-input-9-1bdf1cd647ce>:14: UserWarning: `Model.predict_generator`
is deprecated and will be removed in a future version. Please use `Model.p
redict`, which supports generators.
  predictions = model.predict_generator(test_generator, steps=test_generat
or.samples/test_generator.batch_size,verbose=1)
546/546 [==============================] - 6s 9ms/step
No of errors = 238/546
```

In [ ]:
```
accuracy = ((test_generator.samples-len(errors))/test_generator.samples)
accuracy
```

Out[ ]: 56.41025641025641

In [ ]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
cm = confusion_matrix(y_true=ground_truth, y_pred=predicted_classes)
cm = np.array(cm)
# Normalise
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=label2index, yticklab
plt.ylabel('Actual', fontsize=15)
plt.xlabel('Predicted', fontsize=15)
plt.show(block=False)
```



In [ ]:
```
from sklearn.metrics import classification_report
print(classification_report(ground_truth, predicted_classes, target_names
```

	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.75	0.55	0.64	103
Apple___Apple_scab	1.00	0.01	0.03	134
Apple___Cedar_apple_rust	1.00	0.02	0.04	49
Apple___healthy	0.53	0.95	0.68	260
accuracy			0.56	546
macro avg	0.82	0.39	0.35	546
weighted avg	0.73	0.56	0.46	546

RESULT: We have successfully implemented an image classification task using pre-trained models like AlexNet, VGGNet, InceptionNet and ResNet and compare the results.

Experiment : 5

AIM: Design a CNN architecture to implement the image classification task over an image dataset. Perform the Hyper-parameter tuning and record the results.

Importing Necessary libraries

- From keras library we are going to use image preprocessing task, to normalize the image pixel values in between 0 to 1.
- Model is imported to load variuos Neural NEtwrok models such as Sequential.
- We are going to use Stochastic Gradient Descent(SGD) as a optimizer
- Keras layers such as Dense, Flatten, Conv2D and MaxPooling is used to implement the CNN model

In [ ]:
```
from keras import applications
from keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras import backend as k
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, Tenso
```

In [ ]:
```
import numpy as np
import keras
from keras import models
import matplotlib.pyplot as plt
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
from keras.optimizers import SGD
from keras import layers
from keras.layers import  Dense, Flatten, Conv2D, MaxPooling2D
from keras import Input
```

Loading the training and testing data and defining the basic parameters

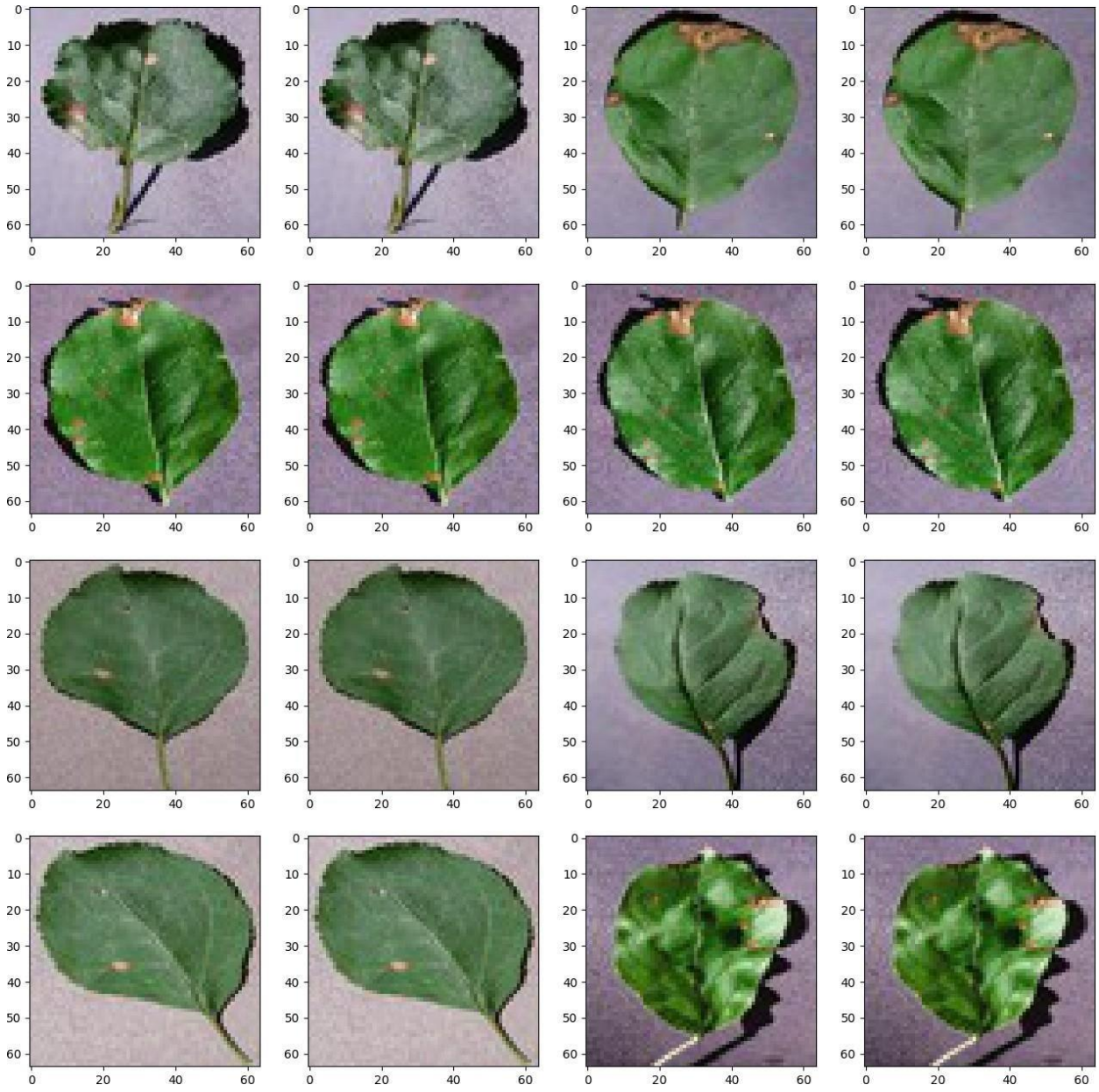
- We are resizing the input image to 128 * 128
- In the dataset : Training Set : 70% Validation Set : 20% Test Set : 10%

In [ ]:
```
# Normalize training and validation data in the range of 0 to 1
train_datagen = ImageDataGenerator(rescale=1./255) # vertical_flip=True,
                                                   # horizontal_flip=True
                                                   # height_shift_range=0
                                                   # width_shift_range=0.
validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Read the training sample and set the batch size
train_generator = train_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(64, 64),
        batch_size=16,
        class_mode='categorical')

# Read Validation data from directory and define target size with batch s
validation_generator = validation_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(64, 64),
        batch_size=16,
        class_mode='categorical',
        shuffle=False)

test_generator = test_datagen.flow_from_directory(
        '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/plant_vi
        target_size=(64, 64),
        batch_size=1,
        class_mode='categorical',
        shuffle=False)
```

```
Found 3002 images belonging to 4 classes.
Found 635 images belonging to 4 classes.
Found 546 images belonging to 4 classes.
```

Visualization of few images

In [ ]:
```
plt.figure(figsize=(16, 16))
for i in range(1, 17):
  plt.subplot(4, 4, i)
  img, label = test_generator.next()
  # print(img.shape)
  # print(label)
  plt.imshow(img[0])
plt.show()
```



In [ ]:
```
img, label = test_generator.next()
img[0].shape
```

Out[ ]: (64, 64, 3)

Model Definition

- We are going to use 2 convolution layers with 3*3 filer and relu as an activation function
- Then max pooling layer with 2*2 filter is used
- After that we are going to use Flatten layer
- Then Dense layer is used with relu function
- In the output layer softmax function is used with 4 neurons as we have four class dataset.

- model.summary() is used to check the overall architecture of the model with number of learnable parameters in each

In [ ]:
```
# Create the model
model = models.Sequential()
# Add new layers
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu', input_shape
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(layers.Flatten())
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(4, activation='softmax'))
model.summary()
```

Compiling and Training the Model

In [ ]:
```
# sgd = SGD(lr=0.001,decay=1e-6, momentum=0.9, nesterov=True)
# We are going to use accuracy metrics and cross entropy loss as performa
model.compile(optimizer = optimizers.Adam(learning_rate = 0.001), loss='c
# Train the model
history = model.fit(train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=validation_generator.samples/validation_generator.
      verbose=1)
```

Saving the model

In [ ]:
```
model.save("CONV_plant_deseas.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

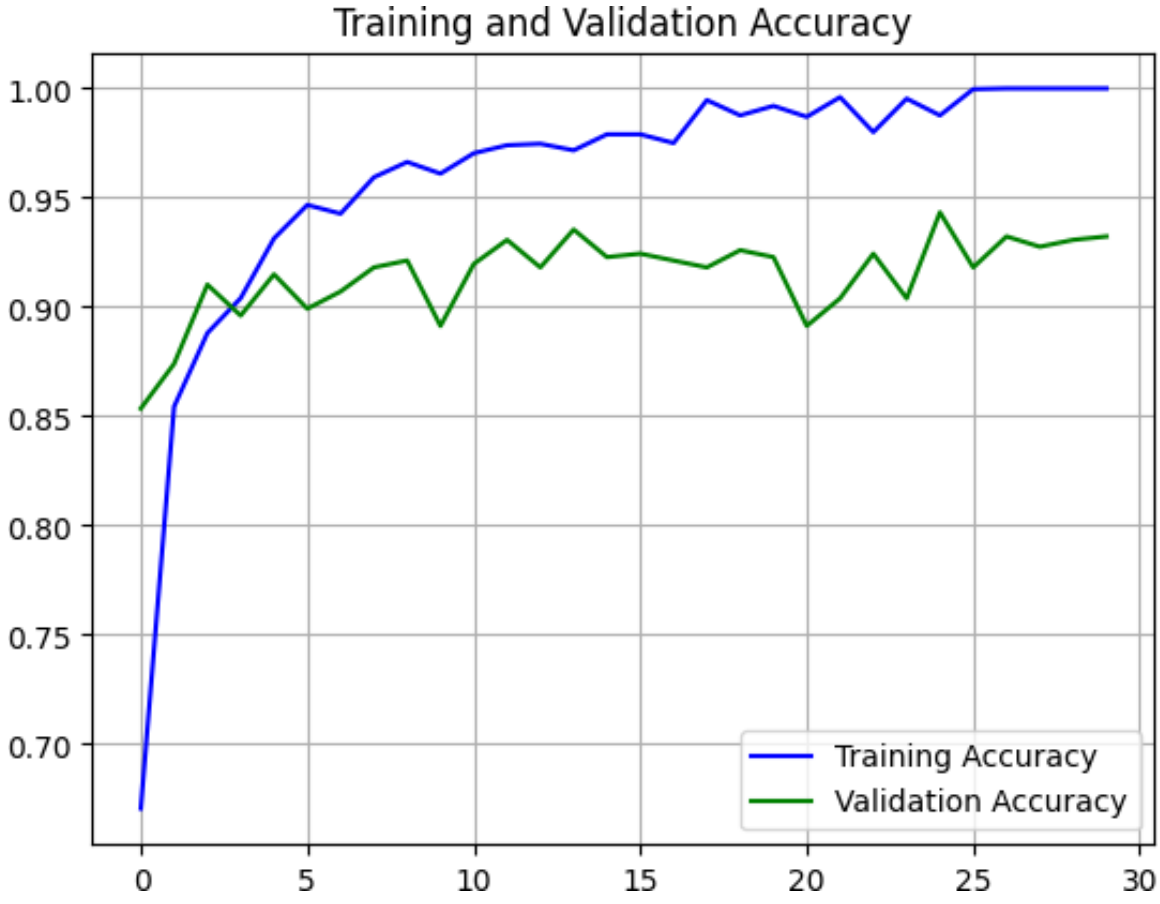
Loading the model

In [ ]:
```
model = models.load_model('CONV_plant_deseas.h5')
```

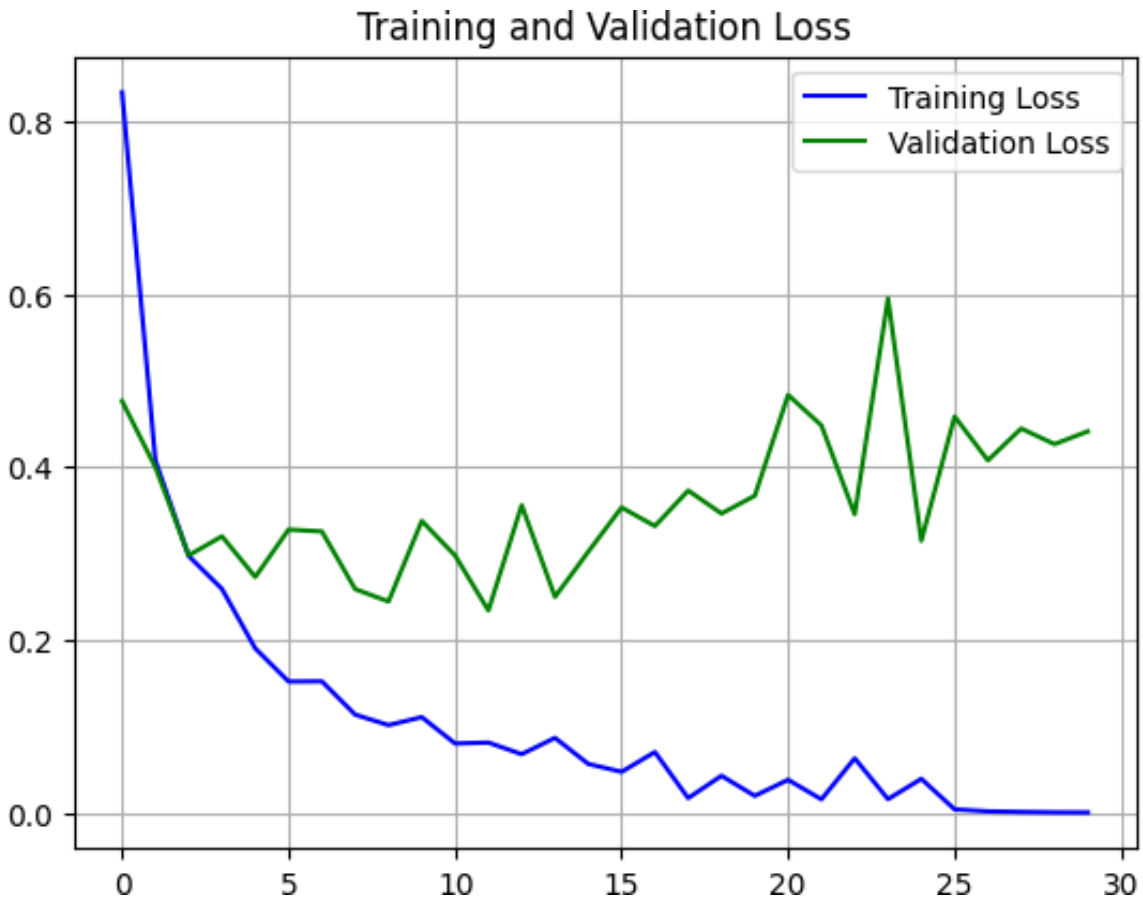
Visualization of Accuracy and Loss Curves

In [ ]:
```
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
```

In [ ]:
```
epochs = range(len(train_acc))
plt.plot(epochs, train_acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.legend()
plt.figure()
plt.show()

plt.plot(epochs, train_loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'g', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.grid()
plt.legend()
plt.show()
```



<Figure size 640x480 with 0 Axes>



Prediction

In [ ]:
```
# Get the filenames from the generator
fnames = test_generator.filenames

# Get the ground truth from generator
ground_truth = test_generator.classes

# Get the label to class mapping from the generator
label2index = test_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(test_generator, steps=test_generato
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),test_generator.samples))
```

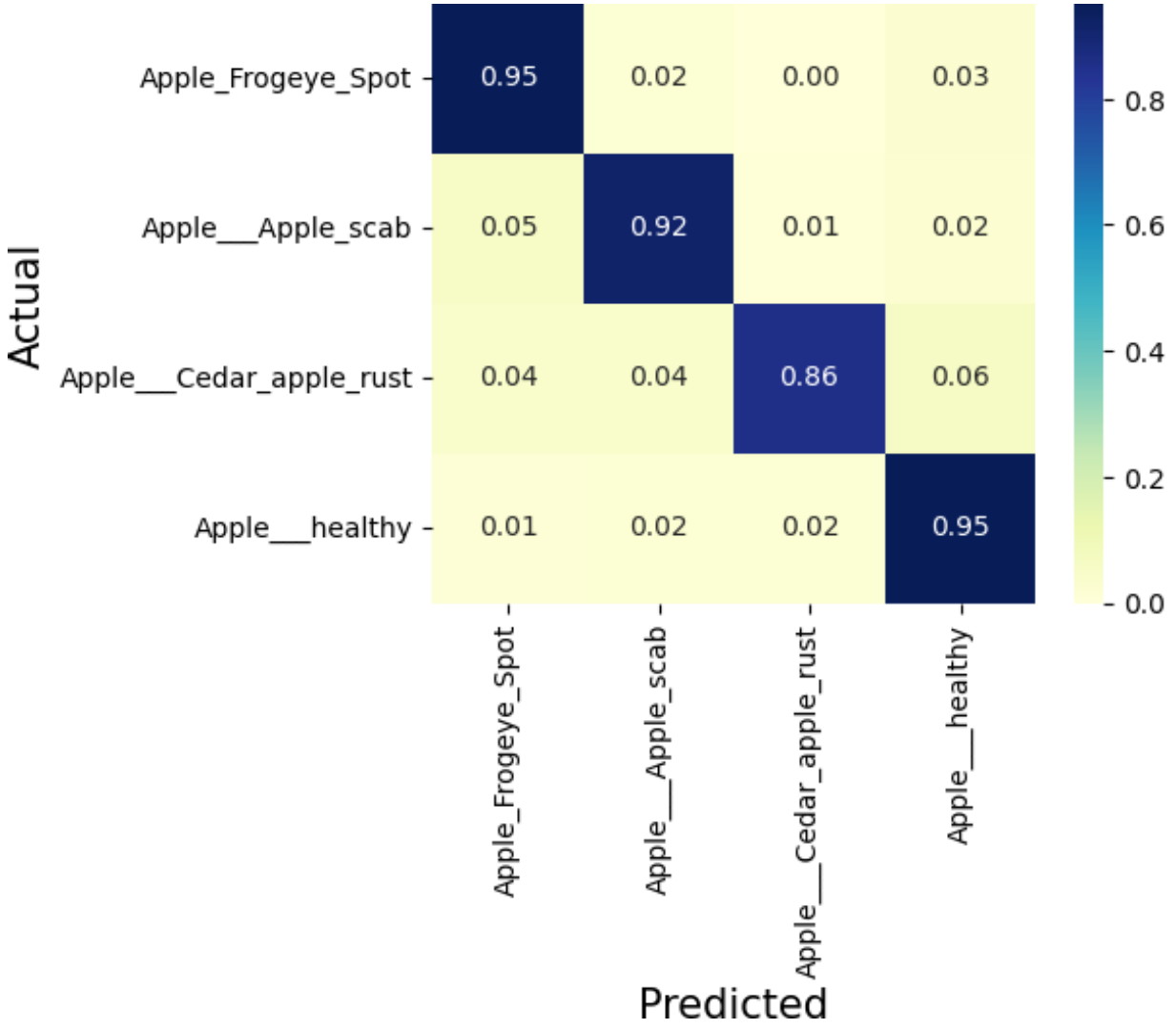
```
<ipython-input-12-1bdf1cd647ce>:14: UserWarning: `Model.predict_generator`
is deprecated and will be removed in a future version. Please use `Model.p
redict`, which supports generators.
  predictions = model.predict_generator(test_generator, steps=test_generat
or.samples/test_generator.batch_size,verbose=1)
546/546 [==============================] - 249s 456ms/step
No of errors = 36/546
```

In [ ]:
```
accuracy = ((test_generator.samples-len(errors))/test_generator.samples)
accuracy
```

Out[ ]: 93.4065934065934

Confusion Matrix

In [ ]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
cm = confusion_matrix(y_true=ground_truth, y_pred=predicted_classes)
cm = np.array(cm)
# Normalise
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=label2index, yticklab
plt.ylabel('Actual', fontsize=15)
plt.xlabel('Predicted', fontsize=15)
plt.show(block=False)
```



Classification Report

In [ ]:
```
from sklearn.metrics import classification_report
print(classification_report(ground_truth, predicted_classes, target_names
```

	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.89	0.95	0.92	103
Apple___Apple_scab	0.93	0.92	0.92	134
Apple___Cedar_apple_rust	0.88	0.86	0.87	49
Apple___healthy	0.96	0.95	0.96	260
accuracy			0.93	546
macro avg	0.92	0.92	0.92	546
weighted avg	0.93	0.93	0.93	546

In [ ]:

Finetuning Hyper-parameters

In [ ]:
```
def create_model(learn_rate=0.01, momentum=0):
    image_size = 128
    input_shape = (image_size, image_size, 3)

    model = models.Sequential()
    model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu', input_s
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dense(4, activation='softmax'))

    # model = Model(input_shape, x)

    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.SGD(lr=learn_rate, momentum=moment
                  metrics=['accuracy'])

    return model
```

In [ ]:
```
learn_rate = [1e-9, 1e-3]
momentum = [0.6, 0.9]


def try_fit(learn_rate,momentum):
    history_page=[]
    for lr in learn_rate:
        for moment in momentum:
            model = create_model(lr,moment)
            history = model.fit_generator(
                train_generator,
                epochs=1,
                validation_data=validation_generator)
            history_page.append(history)
    return history_page

history_page = try_fit(learn_rate,momentum)
history_page[0].history['accuracy']
```

```
<ipython-input-26-491e2ac86ce5>:10: UserWarning: `Model.fit_generator` is
deprecated and will be removed in a future version. Please use `Model.fit
`, which supports generators.
  history = model.fit_generator(
```

```
188/188 [==============================] - 305s 2s/step - loss: 0.6797 - a
ccuracy: 0.4417 - val_loss: 0.6787 - val_accuracy: 0.5039
188/188 [==============================] - 299s 2s/step - loss: 0.7016 - a
ccuracy: 0.2375 - val_loss: 0.7035 - val_accuracy: 0.2000
188/188 [==============================] - 295s 2s/step - loss: 0.5859 - a
ccuracy: 0.3944 - val_loss: 0.4985 - val_accuracy: 0.5181
188/188 [==============================] - 298s 2s/step - loss: 0.5418 - a
ccuracy: 0.4460 - val_loss: 0.4854 - val_accuracy: 0.5181
```

Out[ ]: [0.44170552492141724]

In [ ]:

Experiment : 7

AIM: Train a Recurrent Neural Network (RNN) on the IMDB large movie review dataset for sentiment analysis.

Importing Necessary libraries

In [ ]:
```
import pandas as pd     # to load dataset
import numpy as np      # for mathematic equation
from nltk.corpus import stopwords   # to get collection of stopwords
from sklearn.model_selection import train_test_split       # for splittin
from tensorflow.keras.preprocessing.text import Tokenizer  # to encode te
from tensorflow.keras.preprocessing.sequence import pad_sequences   # to
from tensorflow.keras.models import Sequential      # the model
from tensorflow.keras.layers import Embedding, LSTM, Dense # layers of th
from tensorflow.keras.callbacks import ModelCheckpoint    # save model
from tensorflow.keras.models import load_model    # load saved model
import re
from keras.layers import SimpleRNN
```

Preparing the data named IMDB

In [ ]:
```
data = pd.read_csv('/content/drive/MyDrive/AMITY/Deep Learning (codes)/Da
print(data)
```

Stop Word is a commonly used words in a sentence, usually a search engine is programmed to ignore this words (i.e. "the", "a", "an", "of", etc.) Declaring the english stop words

In [ ]:
```
import nltk
nltk.download("stopwords")
english_stops = set(stopwords.words('english'))
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Load and Clean Dataset

In the original dataset, the reviews are still dirty. There are still html tags,

numbers, uppercase, and punctuations. This will not be good for training, so in load_dataset() function, beside loading the dataset using pandas, I also pre-process the reviews by removing html tags, non alphabet (punctuations and numbers), stop words, and lower case all of the reviews.

Encode Sentiments

In the same function, We also encode the sentiments into integers (0 and 1). Where 0 is for negative sentiments and 1 is for positive sentiments.

In [ ]:
```
def load_dataset():
    df = pd.read_csv('/content/drive/MyDrive/AMITY/Deep Learning (codes)/
    x_data = df['review']       # Reviews/Input
    y_data = df['sentiment']    # Sentiment/Output

    # PRE-PROCESS REVIEW
    x_data = x_data.replace({'<.*?>': ''}, regex = True)          # remov
    x_data = x_data.replace({'[^A-Za-z]': ' '}, regex = True)     # remov
    x_data = x_data.apply(lambda review: [w for w in review.split() if w
    x_data = x_data.apply(lambda review: [w.lower() for w in review])    #

    # ENCODE SENTIMENT -> 0 & 1
    y_data = y_data.replace('positive', 1)
    y_data = y_data.replace('negative', 0)

    return x_data, y_data

x_data, y_data = load_dataset()

print('Reviews')
print(x_data, '\n')
print('Sentiment')
print(y_data)
```

Split Dataset

In this work, We decided to split the data into 80% of Training and 20% of Testing set using train_test_split method from Scikit-Learn. By using this method, it automatically shuffles the dataset. We need to shuffle the data because in the original dataset, the reviews and sentiments are in order, where they list positive reviews first and then negative reviews. By shuffling the data, it will be distributed equally in the model, so it will be more accurate for predictions.

In [ ]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_
```

```
print('Train Set')
print(x_train, '\n')
print(x_test, '\n')
print('Test Set')
print(y_train, '\n')
print(y_test)
```

Function for getting the average review length, by calculating the mean of all the reviews length (using numpy.mean)

In [ ]:
```
def get_max_length():
    review_length =  []
    for review in x_train:
        review_length.append(len(review))

    return int(np.ceil(np.mean(review_length)))
```

Tokenize and Pad/Truncate Reviews

A Neural Network only accepts numeric data, so we need to encode the reviews. I use tensorflow.keras.preprocessing.text.Tokenizer to encode the reviews into integers, where each unique word is automatically indexed (using fit_on_texts method) based on x_train.

x_train and x_test is converted into integers using texts_to_sequences method.

Each reviews has a different length, so we need to add padding (by adding 0) or truncating the words to the same length (in this case, it is the mean of all reviews length) using tensorflow.keras.preprocessing.sequence.pad_sequences.

In [ ]:
```
# ENCODE REVIEW
token = Tokenizer(lower=False)     # no need lower, because already lowere
token.fit_on_texts(x_train)
x_train = token.texts_to_sequences(x_train)
x_test = token.texts_to_sequences(x_test)

max_length = get_max_length()

x_train = pad_sequences(x_train, maxlen=max_length, padding='post', trunc
x_test = pad_sequences(x_test, maxlen=max_length, padding='post', truncat

total_words = len(token.word_index) + 1    # add 1 because of 0 padding
print('Total Words:', total_words)

print('Encoded X Train\n', x_train, '\n')
print('Encoded X Test\n', x_test, '\n')
```

```
print('Maximum review length: ', max_length)
```

Build Architecture/Model

Embedding Layer: in simple terms, it creates word vectors of each word in the word_index and group words that are related or have similar meaning by analyzing other words around them.

RNN Layer: to make a decision to keep or throw away data by considering the current input, previous output.

Dense Layer: compute the input with the weight matrix and bias (optional), and using an activation function. I use Sigmoid activation function for this work because the output is only 0 or 1.

The optimizer is Adam and the loss function is Binary Crossentropy because again the output is only 0 and 1, which is a binary number.

In [ ]:
```
rnn = Sequential()

rnn.add(Embedding(total_words,32,input_length =max_length))
rnn.add(SimpleRNN(64,input_shape = (total_words, max_length), return_sequ
rnn.add(Dense(1, activation = 'sigmoid')) #flatten

print(rnn.summary())
rnn.compile(loss="binary_crossentropy",optimizer='adam',metrics=["accurac
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 130, 32)           2964352

 simple_rnn (SimpleRNN)      (None, 64)                6208

 dense (Dense)               (None, 1)                 65

=================================================================
Total params: 2,970,625
Trainable params: 2,970,625
Non-trainable params: 0
_________________________________________________________________
None
```

Trainin the Model

In [ ]:
```
history = rnn.fit(x_train,y_train,epochs = 20,batch_size=128,verbose = 1)
```

Saving The Model

In [ ]:
```
model = rnn.save('rnn.h5')
loaded_model = load_model('rnn.h5')
```

Evaluation

In [ ]:
```
y_pred = rnn.predict(x_test, batch_size = 128)
print(y_pred)
print(y_test)
for i in range(len(y_pred)):
  if y_pred[i]>0.5:
    y_pred[i] = 1
  else:
    y_pred[i] = 0

true = 0
for i, y in enumerate(y_test):
    if y == y_pred[i]:
        true += 1

print('Correct Prediction: {}'.format(true))
print('Wrong Prediction: {}'.format(len(y_pred) - true))
print('Accuracy: {}'.format(true/len(y_pred)*100))
```

Message: Nothing was typical about this. Everything was beautifully done in this movie, the story, the flow, the scenario, everything. I highly recommend it for mystery lovers, for anyone who wants to watch a good movie!

Example review

In [ ]:
```
review = str(input('Movie Review: '))
```

```
Movie Review: Nothing was typical about this. Everything was beautifully d
one in this movie, the story, the flow, the scenario, everything. I highly
recommend it for mystery lovers, for anyone who wants to watch a good movi
e!
```

Pre-processing of entered review

In [ ]:
```
# Pre-process input
regex = re.compile(r'[^a-zA-Z\s]')
review = regex.sub('', review)
print('Cleaned: ', review)
```

```
words = review.split(' ')
filtered = [w for w in words if w not in english_stops]
filtered = ' '.join(filtered)
filtered = [filtered.lower()]

print('Filtered: ', filtered)
```

```
Cleaned:  Nothing was typical about this Everything was beautifully done i
n this movie the story the flow the scenario everything I highly recommend
it for mystery lovers for anyone who wants to watch a good movie
Filtered: ['nothing typical everything beautifully done movie story flow
scenario everything i highly recommend mystery lovers anyone wants watch g
ood movie']
```

In [ ]:
```
tokenize_words = token.texts_to_sequences(filtered)
tokenize_words = pad_sequences(tokenize_words, maxlen=max_length, padding
print(tokenize_words)
```

```
[[  76  705  174 1210  126    3   13 2692 2596  174    1  442  280  701
  1771  155  400   33    9    3    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0]]
```

Prediction

In [ ]:
```
result = rnn.predict(tokenize_words)
print(result)
```

```
1/1 [==============================] - 0s 40ms/step
[[0.78446704]]
```

In [ ]:
```
if result >= 0.7:
    print('positive')
else:
    print('negative')
```

```
positive
```

RESULT: We have successfully trained a Recurrent Neural Network (RNN) on the IMDB large movie review dataset for sentiment analysis.

Experiment : 8

AIM: Download the tweets data from twitter and run an RNN/LSTM to classify the sentiment as Hate/ Non-hate speech. Compare the results of both the models.

Importing Necessary libraries

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Preparing the data named IMDB

In [ ]:
```
df = pd.read_csv('/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data
df.head()
```

Out[ ]:

	Unnamed: 0	count	hate_speech	offensive_language	neither	class	
0	0	3	0	0	3	2	@mayasol As a woma shoul
1	1	3	0	3	0	1	! @mleew1 dats cold. dw
2	2	3	0	3	0	1	!!! @UrKindOf Dawg @80sb
3	3	3	0	2	1	1	!!!!! @C_G_And @viva_base
4	4	6	0	6	0	1	!!!!!!!!! @ShenikaRo The shit

In [ ]:
```
classes = ['Hate Speech','Offensive Language','None']
```

In [ ]:
```
df.drop(['count','hate_speech','offensive_language','neither','Unnamed: 0
```

In [ ]:
```
df.head()
```

Out[ ]:

	class	tweet
0	2	!!! RT @mayasolovely: As a woman you shouldn't...
1	1	!!!!! RT @mleew17: boy dats cold...tyga dwn ba...
2	1	!!!!!!! RT @UrKindOfBrand Dawg!!!! RT @80sbaby...
3	1	!!!!!!!!! RT @C_G_Anderson: @viva_based she lo...
4	1	!!!!!!!!!!!!! RT @ShenikaRoberts: The shit you...

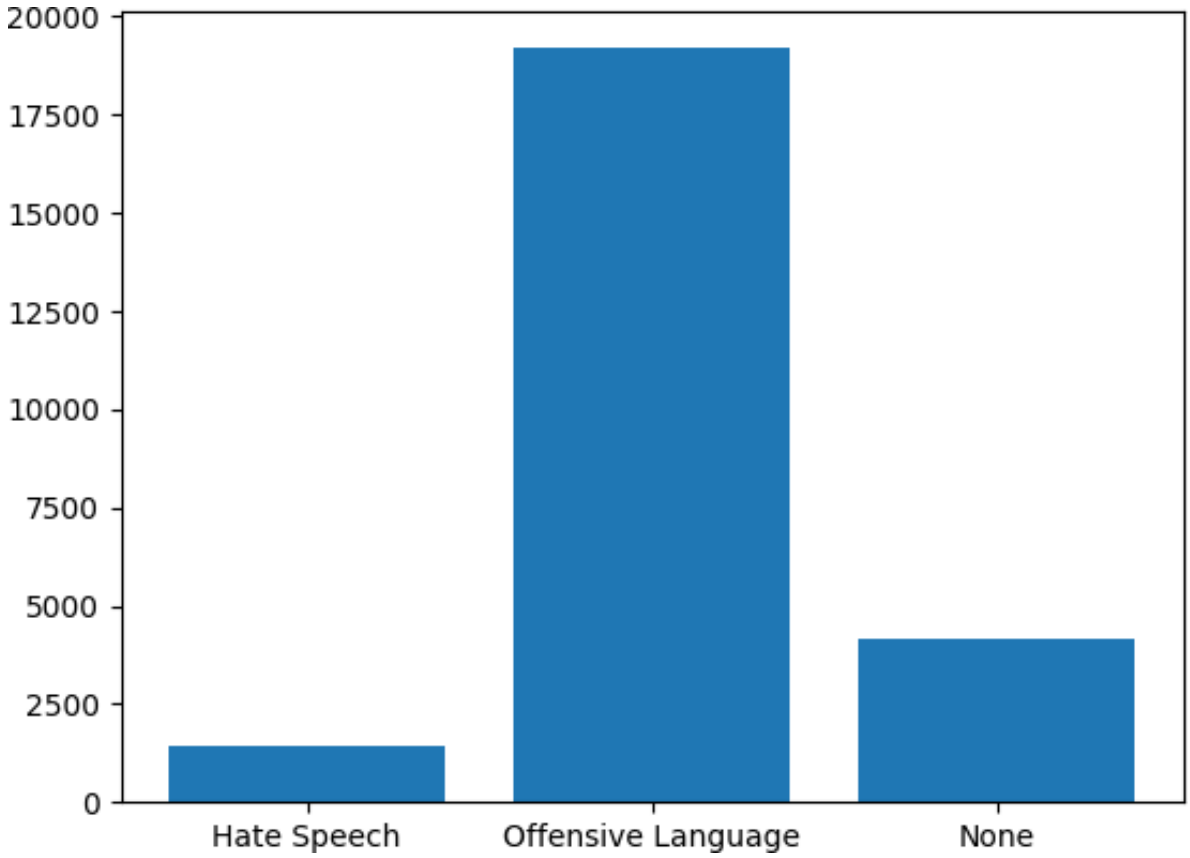
In [ ]:
```
df.shape
```

Out[ ]:
```
(24783, 2)
```

Statistics of the Data

In [ ]:
```
labels = df['class']
unique, counts = np.unique(labels, return_counts=True)
values = list(zip(unique, counts))
plt.bar(classes,counts)
for i in values:
    print(classes[i[0]],' : ',i[1])
plt.show()
```

```
Hate Speech :  1430
Offensive Language  :  19190
None  :  4163
```



In [ ]:
```
hate_tweets = df[df['class']==0]
offensive_tweets = df[df['class']==1]
neither = df[df['class']==2]
print(hate_tweets.shape)
print(offensive_tweets.shape)
print(neither.shape)
```

```
(1430, 2)
(19190, 2)
(4163, 2)
```

In [ ]:
```
for i in range(3):
    hate_tweets = pd.concat([hate_tweets,hate_tweets],ignore_index = True
neither = pd.concat([neither,neither,neither], ignore_index = True)
offensive_tweets = offensive_tweets.iloc[0:12000,:]
print(hate_tweets.shape)
print(offensive_tweets.shape)
print(neither.shape)
```

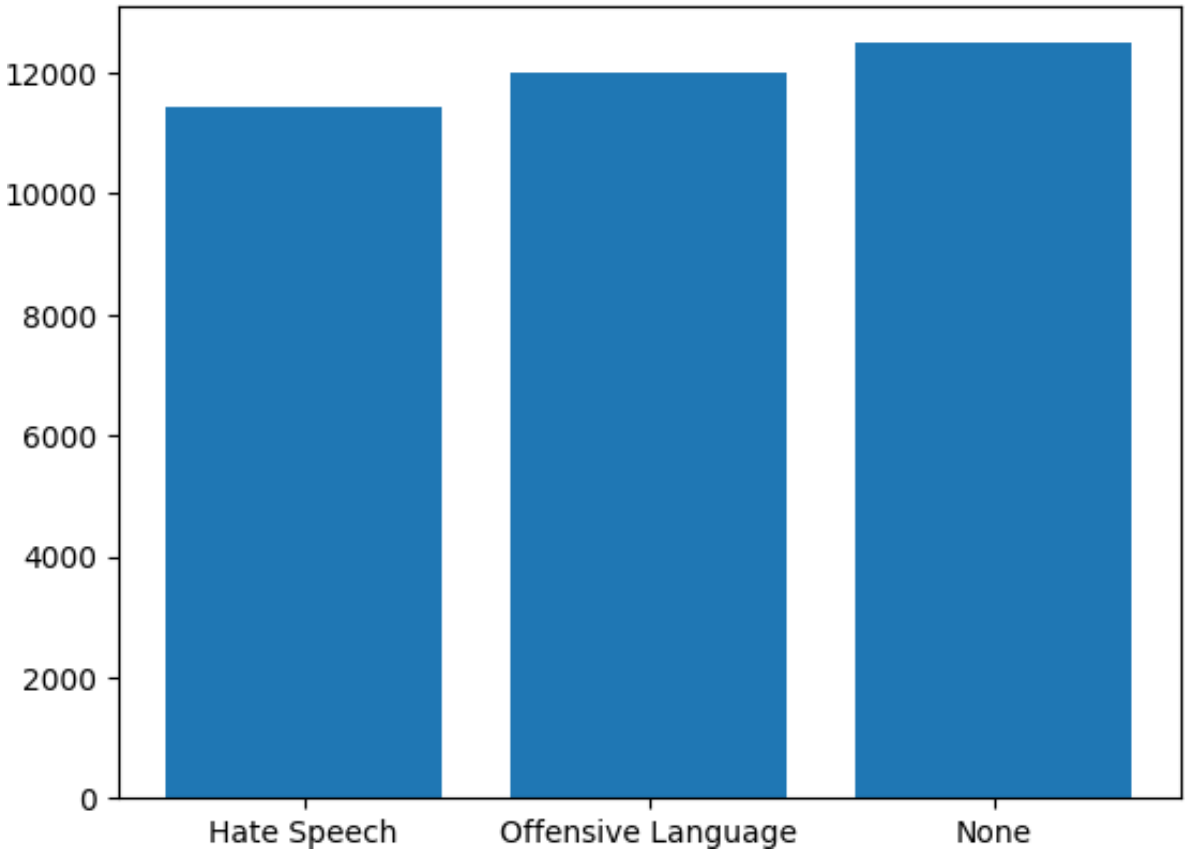
```
(11440, 2)
(12000, 2)
(12489, 2)
```

In [ ]:
```
df = pd.concat([hate_tweets,offensive_tweets,neither],ignore_index = True
df.shape
```

Out[ ]: (35929, 2)

In [ ]:
```
labels = df['class']
unique, counts = np.unique(labels, return_counts=True)
values = list(zip(unique, counts))
plt.bar(classes,counts)
for i in values:
    print(classes[i[0]],' : ',i[1])
plt.show()
```

```
Hate Speech : 11440
Offensive Language  :  12000
None  :  12489
```



In [ ]:
```
df.head()
```

Out[ ]:

	class	tweet
0	0	"@Blackman38Tide: @WhaleLookyHere @HowdyDowdy1...
1	0	"@CB_Baby24: @white_thunduh alsarabsss" hes a ...
2	0	"@DevilGrimz: @VigxRArts you're fucking gay, b...
3	0	"@MarkRoundtreeJr: LMFAOOOO I HATE BLACK PEOPL...
4	0	"@NoChillPaz: "At least I'm not a nigger" http...

Preprocessing

In [ ]:
```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import re
nltk.download('wordnet')
nltk.download('stopwords')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Out[ ]: True

In [ ]:
```
# dealing with Slangs
d = {'luv':'love','wud':'would','lyk':'like','wateva':'whatever','ttyl':'
                'kul':'cool','fyn':'fine','omg':'oh my god!','fam':'family
                'cud':'could','fud':'food', 'u': 'you',
     'ur':'your', 'bday' : 'birthday', 'bihday' : 'birthday'}
```

In [ ]:
```
stop_words = set(stopwords.words("english"))
stop_words.add('rt')
stop_words.remove('not')
lemmatizer = WordNetLemmatizer()
giant_url_regex = ('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|' '[!*\(\),]|(
mention_regex = '@[\w\-]+'

def clean_text(text):
    text = re.sub('"', "", text)
    text = re.sub(mention_regex, ' ',text) #removing all user names
    text = re.sub(giant_url_regex, ' ', text) #remocing the urls
    text = text.lower()
    text = re.sub("hm+", "", text) #removing variants of hmmm
    text = re.sub("[^a-z]+", " ", text) #removing all numbers, special ch
    text = text.split()
    text = [word for word in text if not word in stop_words]
    text = [d[word] if word in d else word for word in text]  #replacing
    text = [lemmatizer.lemmatize(token) for token in text]
    text = [lemmatizer.lemmatize(token, "v") for token in text]
    text = " ".join(text)
    return text
```

In [ ]:
```
df['processed_tweets'] = df.tweet.apply(lambda x: clean_text(x))    # df.r
df.head()
```

Out[ ]:

	class	tweet	processed_tweets
0	0	"@Blackman38Tide: @WhaleLookyHere @HowdyDowdy1...	queer gaywad
1	0	"@CB_Baby24: @white_thunduh alsarabsss" hes a ...	alsarabsss he beaner smh tell he mexican
2	0	"@DevilGrimz: @VigxRArts you're fucking gay, b...	fuck gay blacklist hoe hold tehgodclan anyway
3	0	"@MarkRoundtreeJr: LMFAOOOO I HATE BLACK PEOPL...	lmfaoooo hate black people black people nigger
4	0	"@NoChillPaz: "At least I'm not a nigger" http...	least not nigger lmfao

In [ ]:
```
x = df.processed_tweets
y = df['class']
print(x.shape)
print(y.shape)
```

```
(35929,)
(35929,)
```

In [ ]:
```
# finding unique words
word_unique = []
for i in x:
    for j in i.split():
        word_unique.append(j)
unique, counts = np.unique(word_unique, return_counts=True)
print("The total words in the tweets are : ", len(word_unique))
print("The total UNIQUE words in the tweets are : ", len(unique))
```

```
The total words in the tweets are :  275540
The total UNIQUE words in the tweets are :  14146
```

In [ ]:
```
# finding length of tweets
tweets_length = []
for i in x:
    tweets_length.append(len(i.split()))
print("The Average Length tweets are : ",np.mean(tweets_length))
print("The max length of tweets is : ", np.max(tweets_length))
print("The min length of tweets is : ", np.min(tweets_length))
```

```
The Average Length tweets are :  7.669013888502324
The max length of tweets is :  28
The min length of tweets is :  0
```

In [ ]:
```
tweets_length = pd.DataFrame(tweets_length)
# tweets_length.describe()
```

Out[ ]:

	0
count	35929.000000
mean	7.669014
std	3.989625
min	0.000000
25%	4.000000
50%	7.000000
75%	11.000000
max	28.000000

In [ ]:
```
#Sorting the Unique words based on their Frequency
col = list(zip(unique, counts))
col = sorted(col, key = lambda x: x[1],reverse=True)
col=pd.DataFrame(col)
print("Top 20 Occuring Words with their frequency are:")
col.iloc[:20,:]
```

```
Top 20 Occuring Words with their frequency are:
```

Out[ ]:

	0	1
0	bitch	9066
1	like	3817
2	get	3636
3	hoe	3426
4	trash	3217
5	fuck	3103
6	nigga	2819
7	faggot	2239
8	as	2073
9	you	1851
10	pussy	1847
11	go	1773
12	not	1764
13	bird	1515
14	lol	1494
15	nigger	1459
16	say	1456
17	make	1373
18	amp	1329
19	white	1328

In [ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

In [ ]:
```
vectorizer = TfidfVectorizer(max_features = 8000 )
# tokenize and build vocab

vectorizer.fit(x)
# summarize

print(len(vectorizer.vocabulary_))
print(vectorizer.idf_.shape)
```

```
8000
(8000,)
```

In [ ]:
```
x_tfidf = vectorizer.transform(x).toarray()
print(x_tfidf.shape)
```

```
(35929, 8000)
```

In [ ]:
```
from keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

In [ ]:
```
num_words = 8000
embed_dim = 32
tokenizer = Tokenizer(num_words=num_words,oov_token = "<oov>" )
tokenizer.fit_on_texts(x)
word_index=tokenizer.word_index
sequences = tokenizer.texts_to_sequences(x)
length=[]
for i in sequences:
    length.append(len(i))
print(len(length))
print("Mean is: ",np.mean(length))
print("Max is: ",np.max(length))
print("Min is: ",np.min(length))
```

```
35929
Mean is:  7.669013888502324
Max is:  28
Min is:  0
```

In [ ]:
```
pad_length = 24
sequences = pad_sequences(sequences, maxlen = pad_length, truncating = 'p
sequences.shape
```

Out[ ]: (35929, 24)

Splitting the Data

In [ ]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(sequences,y,test_size =
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(34132, 24)
(1797, 24)
(34132,)
(1797,)
```

RNN Model

In [ ]:
```
from keras.layers import Dense, Embedding, Dropout , Activation, Flatten,
from keras.layers import GlobalMaxPool1D
from keras.models import Model, Sequential
```

```
import tensorflow as tf
```

In [ ]:
```
recall = tf.keras.metrics.Recall()
precision = tf.keras.metrics.Precision()

model = Sequential([Embedding(num_words, embed_dim, input_length = pad_le
                    SimpleRNN(8, return_sequences = True),
                    GlobalMaxPool1D(),
                    Dense(20,activation = 'relu',kernel_initializer='he_un
                    Dropout(0.25),
                    Dense(3,activation = 'softmax')])
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', m
# model.name = 'Twitter Hate Text Classification'
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 24, 32)            256000

 simple_rnn_1 (SimpleRNN)    (None, 24, 8)             328

 global_max_pooling1d_1 (Glo  (None, 8)                0
 balMaxPooling1D)

 dense_2 (Dense)             (None, 20)                180

 dropout_1 (Dropout)         (None, 20)                0

 dense_3 (Dense)             (None, 3)                 63

=================================================================
Total params: 256,571
Trainable params: 256,571
Non-trainable params: 0
_________________________________________________________________
```

Training the Model

In [ ]:
```
history = model.fit(x = x_train, y = y_train, epochs = 5,validation_split
```

```
Epoch 1/5
1014/1014 [==============================] - 12s 12ms/step - loss: 0.2029
- accuracy: 0.9373 - val_loss: 0.1583 - val_accuracy: 0.9473
Epoch 2/5
1014/1014 [==============================] - 11s 11ms/step - loss: 0.1271
- accuracy: 0.9624 - val_loss: 0.1521 - val_accuracy: 0.9467
Epoch 3/5
1014/1014 [==============================] - 12s 11ms/step - loss: 0.0906
- accuracy: 0.9754 - val_loss: 0.1479 - val_accuracy: 0.9537
Epoch 4/5
1014/1014 [==============================] - 12s 12ms/step - loss: 0.0714
- accuracy: 0.9801 - val_loss: 0.1644 - val_accuracy: 0.9525
Epoch 5/5
1014/1014 [==============================] - 12s 12ms/step - loss: 0.0588
- accuracy: 0.9839 - val_loss: 0.1541 - val_accuracy: 0.9584
```

Evaluation

In [ ]:
```
evaluate = model.evaluate(x_test,y_test)
```

```
57/57 [==============================] - 0s 3ms/step - loss: 0.1112 - accu
racy: 0.9688
```

In [ ]:
```
print("Test Acuracy is : {:.2f} %".format(evaluate[1]*100))
print("Test Loss is : {:.4f}".format(evaluate[0]))
```

```
Test Acuracy is : 96.88 %
Test Loss is : 0.1112
```

In [ ]:
```
predictions = model.predict(x_test)
```

```
57/57 [==============================] - 0s 2ms/step
```

In [ ]:
```
predict = []
for i in predictions:
    predict.append(np.argmax(i))
```

Confusion Matrix

In [ ]:
```
from sklearn import metrics
cm = metrics.confusion_matrix(predict,y_test)
acc = metrics.accuracy_score(predict,y_test)
```

In [ ]:
```
print("The Confusion matrix is: \n",cm)
```

```
The Confusion matrix is:
 [[548  22   1]
 [  6 572   8]
 [  0  19 621]]
```

Accuracy

In [ ]:
```
print(acc*100)
```

```
96.88369504730106
```

Classification Report

In [ ]:
```
from sklearn import metrics
print(metrics.classification_report(y_test, predict))
```

```
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       554
           1       0.98      0.93      0.95       613
           2       0.97      0.99      0.98       630

    accuracy                           0.97      1797
   macro avg       0.97      0.97      0.97      1797
weighted avg       0.97      0.97      0.97      1797
```

LSTM

In [ ]:
```
from tensorflow.keras.layers import Embedding, LSTM, Dense
# ARCHITECTURE
EMBED_DIM = 32
LSTM_OUT = 64

model = Sequential()
model.add(Embedding(num_words, EMBED_DIM, input_length = pad_length))
model.add(LSTM(LSTM_OUT))
model.add(Dense(3, activation='softmax'))
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy

print(model.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, 24, 32)            256000

 lstm_1 (LSTM)               (None, 64)                24832

 dense_5 (Dense)             (None, 3)                 195

=================================================================
Total params: 281,027
Trainable params: 281,027
Non-trainable params: 0
_________________________________________________________________
None
```

Training Model

In [ ]:
```
history = model.fit(x = x_train, y = y_train, epochs = 10,validation_spli
```

```
Epoch 1/10
1014/1014 [==============================] - 26s 24ms/step - loss: 0.4328
- accuracy: 0.8246 - val_loss: 0.2531 - val_accuracy: 0.9127
Epoch 2/10
1014/1014 [==============================] - 22s 22ms/step - loss: 0.1874
- accuracy: 0.9405 - val_loss: 0.1876 - val_accuracy: 0.9367
Epoch 3/10
1014/1014 [==============================] - 23s 23ms/step - loss: 0.1426
- accuracy: 0.9554 - val_loss: 0.1991 - val_accuracy: 0.9391
Epoch 4/10
1014/1014 [==============================] - 24s 23ms/step - loss: 0.1188
- accuracy: 0.9626 - val_loss: 0.1991 - val_accuracy: 0.9361
Epoch 5/10
1014/1014 [==============================] - 23s 23ms/step - loss: 0.0999
- accuracy: 0.9688 - val_loss: 0.2142 - val_accuracy: 0.9408
Epoch 6/10
1014/1014 [==============================] - 23s 22ms/step - loss: 0.0835
- accuracy: 0.9730 - val_loss: 0.1820 - val_accuracy: 0.9508
Epoch 7/10
1014/1014 [==============================] - 24s 23ms/step - loss: 0.0735
- accuracy: 0.9767 - val_loss: 0.2202 - val_accuracy: 0.9473
Epoch 8/10
1014/1014 [==============================] - 24s 23ms/step - loss: 0.0670
- accuracy: 0.9783 - val_loss: 0.2666 - val_accuracy: 0.9461
Epoch 9/10
1014/1014 [==============================] - 22s 22ms/step - loss: 0.0577
- accuracy: 0.9815 - val_loss: 0.1988 - val_accuracy: 0.9490
Epoch 10/10
1014/1014 [==============================] - 24s 23ms/step - loss: 0.0537
- accuracy: 0.9823 - val_loss: 0.2313 - val_accuracy: 0.9490
```

Evaluation

In [ ]:
```
evaluate = model.evaluate(x_test,y_test)
```

```
57/57 [==============================] - 0s 8ms/step - loss: 0.1719 - accu
racy: 0.9594
```

In [ ]:
```
print("Test Acuracy is : {:.2f} %".format(evaluate[1]*100))
print("Test Loss is : {:.4f}".format(evaluate[0]))
```

```
Test Acuracy is : 95.94 %
Test Loss is : 0.1719
```

In [ ]:
```
predictions = model.predict(x_test)
```

```
57/57 [==============================] - 1s 8ms/step
```

In [ ]:
```
predict = []
for i in predictions:
    predict.append(np.argmax(i))
```

Confusion Matrix

In [ ]:
```
from sklearn import metrics
cm = metrics.confusion_matrix(predict,y_test)
acc = metrics.accuracy_score(predict,y_test)
```

In [ ]:
```
print("The Confusion matrix is: \n",cm)
```

```
The Confusion matrix is:
 [[552  42   1]
 [  0 543   0]
 [  2  28 629]]
```

Accuracy

In [ ]:
```
print(acc*100)
```

```
95.93767390094602
```

Classification Report

In [ ]:
```
from sklearn import metrics
print(metrics.classification_report(y_test, predict))
```

	precision	recall	f1-score	support
0	0.93	1.00	0.96	554
1	1.00	0.89	0.94	613
2	0.95	1.00	0.98	630
accuracy			0.96	1797
macro avg	0.96	0.96	0.96	1797
weighted avg	0.96	0.96	0.96	1797

In [ ]:

Experiment:-9

AIM:- Implement GAN architecture on MNIST dataset to recognize the handwritten digits.

Importing Necessary libraries

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
```

Pre-processing

Normalizing the data into 0 to 1 and reshaping the size

```
def preprocess(array):
    """
    Normalizes the supplied array and reshapes it into the appropriate format.
    """

    array = array.astype("float32") / 255.0
    array = np.reshape(array, (len(array), 28, 28, 1))
    return array
```

Adding noise to the original images

```
def noise(array):
    """
    Adds random noise to each image in the supplied array.

    noise_factor = 0.4
    noisy_array = array + noise_factor * np.random.normal(
        loc=0.0, scale=1.0, size=array.shape
    )

    return np.clip(noisy_array, 0.0, 1.0)
```

Visualizing the images

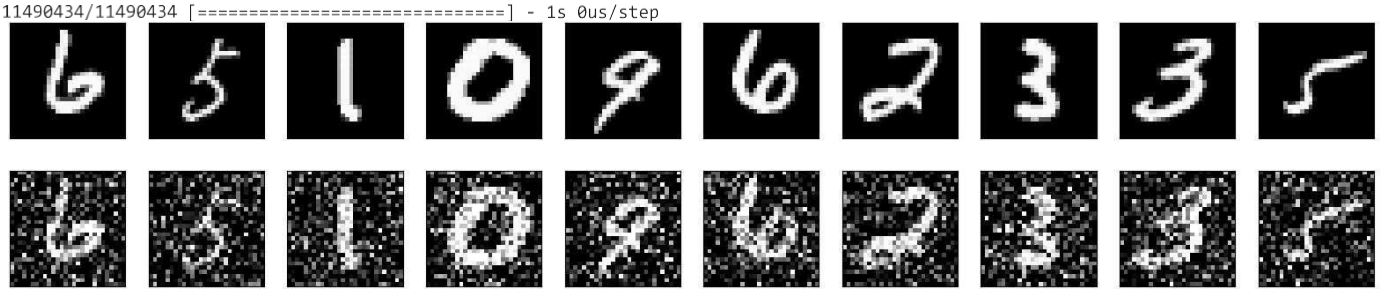
```
def display(array1, array2):
    """
    Displays ten random images from each one of the supplied arrays.
    """

    n = 10

    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]
    images2 = array2[indices, :]

    plt.figure(figsize=(20, 4))
    for i, (image1, image2) in enumerate(zip(images1, images2)):
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(image1.reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(image2.reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```

```
    plt.show()
```

Preparing the data

```
# Since we only need images from the dataset to encode and decode,  we
# won't use the labels.
(train_data, _), (test_data, _) = mnist.load_data()

# Normalize and reshape the data
train_data = preprocess(train_data)
test_data = preprocess(test_data)

# Create a copy of the data with added noise
noisy_train_data = noise(train_data)
noisy_test_data = noise(test_data)

# Display the train data and a version of it with  added noise
display(train_data, noisy_train_data)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
```



Building the Autoencoder

```
input = layers.Input(shape=(28, 28, 1))

# Encoder
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(input)
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(x)
x = layers.MaxPooling2D((2, 2), padding="same")(x)

# Decoder
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(1, (3, 3), activation="sigmoid", padding="same")(x)

# Autoencoder
autoencoder = Model(input, x)
autoencoder.compile(optimizer="adam", loss="binary_crossentropy")
autoencoder.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 32)        9248
```

```
 max_pooling2d_1 (MaxPooling  (None, 7, 7, 32)         0
 2D)

 conv2d_transpose (Conv2DTra  (None, 14, 14, 32)       9248
 nspose)

 conv2d_transpose_1 (Conv2DT  (None, 28, 28, 32)       9248
 ranspose)

 conv2d_2 (Conv2D)           (None, 28, 28, 1)         289

=================================================================
Total params: 28,353
Trainable params: 28,353
Non-trainable params: 0
_________________________________________________________________
```

Training the model

```
autoencoder.fit(
    x=train_data,
    y=train_data,
    epochs=50,
    batch_size=128,
    shuffle=True,
    validation_data=(test_data, test_data),
)
```

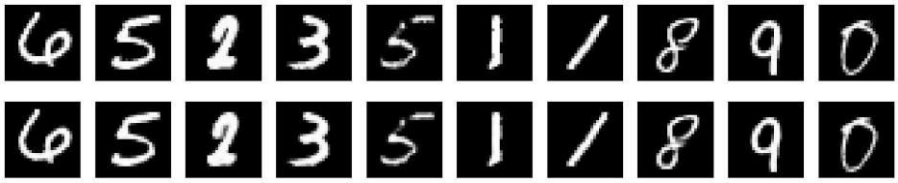
```
Epoch 1/50
469/469 [==============================] - 16s 11ms/step - loss: 0.1303 - val_loss: 0.0732
Epoch 2/50
469/469 [==============================]
Epoch 3/50                               - 4s 8ms/step - loss: 0.0718 - val_loss: 0.0699
469/469 [==============================]
Epoch 4/50                               - 4s 8ms/step - loss: 0.0695 - val_loss: 0.0684
469/469 [==============================]
Epoch 5/50                               - 4s 9ms/step - loss: 0.0683 - val_loss: 0.0674
469/469 [==============================]
Epoch 6/50                               - 4s 8ms/step - loss: 0.0675 - val_loss: 0.0666
469/469 [==============================]
Epoch 7/50                               - 4s 8ms/step - loss: 0.0669 - val_loss: 0.0661
469/469 [==============================]
Epoch 8/50                               - 4s 9ms/step - loss: 0.0664 - val_loss: 0.0659
469/469 [==============================]
Epoch 9/50                               - 4s 8ms/step - loss: 0.0661 - val_loss: 0.0654
469/469 [==============================]
Epoch 10/50                              - 4s 8ms/step - loss: 0.0657 - val_loss: 0.0652
469/469 [==============================]
Epoch 11/50                              - 4s 9ms/step - loss: 0.0655 - val_loss: 0.0649
469/469 [==============================]
Epoch 12/50                              - 4s 8ms/step - loss: 0.0652 - val_loss: 0.0647
469/469 [==============================]
Epoch 13/50                              - 4s 8ms/step - loss: 0.0650 - val_loss: 0.0646
469/469 [==============================]
Epoch 14/50                              - 4s 9ms/step - loss: 0.0648 - val_loss: 0.0643
469/469 [==============================]
Epoch 15/50                              - 4s 8ms/step - loss: 0.0647 - val_loss: 0.0642
469/469 [==============================]
Epoch 16/50                              - 4s 8ms/step - loss: 0.0645 - val_loss: 0.0641
469/469 [==============================]
Epoch 17/50                              - 4s 9ms/step - loss: 0.0644 - val_loss: 0.0640
469/469 [==============================]
Epoch 18/50                              - 4s 8ms/step - loss: 0.0642 - val_loss: 0.0638
469/469 [==============================]
Epoch 19/50                              - 4s 8ms/step - loss: 0.0641 - val_loss: 0.0637
469/469 [==============================]
Epoch 20/50                              - 4s 9ms/step - loss: 0.0640 - val_loss: 0.0636
469/469 [==============================]
Epoch 21/50                              - 4s 8ms/step - loss: 0.0639 - val_loss: 0.0636
469/469 [==============================]
Epoch 22/50                              - 4s 8ms/step - loss: 0.0638 - val_loss: 0.0634
469/469 [==============================]
Epoch 23/50                              - 4s 9ms/step - loss: 0.0637 - val_loss: 0.0633
469/469 [==============================]
Epoch 24/50                              - 4s 8ms/step - loss: 0.0636 - val_loss: 0.0634
469/469 [==============================]
Epoch 25/50                              - 4s 8ms/step - loss: 0.0636 - val_loss: 0.0633
469/469 [==============================]
Epoch 26/50                              - 4s 9ms/step - loss: 0.0635 - val_loss: 0.0631
469/469 [==============================]
                                         - 4s 8ms/step - loss: 0.0634 - val_loss: 0.0630
```

```
Epoch 27/50
469/469 [==============================] - 4s 8ms/step - loss: 0.0634 - val_loss: 0.0630
Epoch 28/50
469/469 [==============================] - 4s 9ms/step - loss: 0.0633 - val_loss: 0.0629
Epoch 29/50
469/469 [==============================] - 4s 8ms/step - loss: 0.0632 - val_loss: 0.0628
```

Prediction

```
predictions = autoencoder.predict(test_data)
display(test_data, predictions)
```

```
313/313 [==============================] - 1s 2ms/step
```



```
autoencoder.fit(
    x=noisy_train_data,
    y=train_data,
    epochs=100,
    batch_size=128,
    shuffle=True,
    validation_data=(noisy_test_data, test_data),
)
```

```
Epoch 1/100
469/469 [==============================] - 4s 9ms/step - loss: 0.1023 - val_loss: 0.0943
Epoch 2/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0937 - val_loss: 0.0919
Epoch 3/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0920 - val_loss: 0.0906
Epoch 4/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0908 - val_loss: 0.0896
Epoch 5/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0900 - val_loss: 0.0891
Epoch 6/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0894 - val_loss: 0.0884
Epoch 7/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0889 - val_loss: 0.0880
Epoch 8/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0885 - val_loss: 0.0877
Epoch 9/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0881 - val_loss: 0.0874
Epoch 10/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0878 - val_loss: 0.0871
Epoch 11/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0875 - val_loss: 0.0868
Epoch 12/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0873 - val_loss: 0.0866
Epoch 13/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0871 - val_loss: 0.0866
Epoch 14/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0869 - val_loss: 0.0863
Epoch 15/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0868 - val_loss: 0.0863
Epoch 16/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0866 - val_loss: 0.0861
Epoch 17/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0865 - val_loss: 0.0860
Epoch 18/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0864 - val_loss: 0.0859
Epoch 19/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0863 - val_loss: 0.0858
Epoch 20/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0862 - val_loss: 0.0857
Epoch 21/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0861 - val_loss: 0.0857
Epoch 22/100
```

```
469/469 [==============================] - 4s 9ms/step - loss: 0.0860 - val_loss: 0.0855
Epoch 23/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0859 - val_loss: 0.0854
Epoch 24/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0858 - val_loss: 0.0855
Epoch 25/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0858 - val_loss: 0.0856
Epoch 26/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0857 - val_loss: 0.0852
Epoch 27/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0856 - val_loss: 0.0852
Epoch 28/100
469/469 [==============================] - 4s 9ms/step - loss: 0.0856 - val_loss: 0.0851
Epoch 29/100
469/469 [==============================] - 4s 8ms/step - loss: 0.0855 - val_loss: 0.0851
```

```
predictions = autoencoder.predict(noisy_test_data)
display(noisy_test_data, predictions)
```

```
313/313 [==============================] - 1s 2ms/step
```