



# SRI RAMACHANDRA

INSTITUTE OF HIGHER EDUCATION AND RESEARCH  
(Category - I Deemed to be University) Porur, Chennai  
SRI RAMACHANDRA ENGINEERING AND TECHNOLOGY

## Title of the project:

Animals Classifier

## Team Members:

Nithyashree R (E0119038)

Vidarshana G (E0119067)

Sajeth S A (E0119027)

## Problem Statement:

- To implement CO1: Performance enhancement - Ensemble techniques
- To implement CO2: Tree Based Learning Methods
- To implement CO3: Dimensionality Reduction
- To implement CO4: Artificial Neural Network
- To implement CO5: Case Study on Model Selection

## Data Description :

The “Animals” dataset is a simple example dataset I put together to demonstrate how to train image classifiers using simple machine learning techniques as well as advanced deep learning algorithms.

Images inside the Animals dataset belong to three distinct classes: dogs, cats, and pandas as you can see in Figure 1, with 1,000 example images per class. The dog and cat images were sampled from the Kaggle Dogs vs. Cats challenge, while the panda images were sampled from the ImageNet dataset. But the whole dataset can be sourced from kaggle <https://www.kaggle.com/ashishsaxena2209/animal-image-datasetdog-cat-and-panda>

## Software Requirements and Platforms used:

Python3 - for developing the model and implementation

Google Colab - Environment used to execute the Python code.

In [ ]:

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

Mounted at /content/drive

In [ ]:

```
cd '/content/drive/MyDrive/CSE - 340/animals'
```

/content/drive/MyDrive/CSE - 340/animals

## Importing the necessary libraries and packages

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import SGD
import numpy as np
from keras.models import load_model
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from imutils import paths
import argparse
from simplepreprocessor import SimplePreprocessor
from simpledatasetloader import SimpleDatasetLoader
import pickle
import os
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
```

## Function to get all image files in a list

In [ ]:

```
def getListOfFiles(dirName):
    listOfFile = os.listdir(dirName)
    allFiles = list()
    for entry in listOfFile:
        fullPath = os.path.join(dirName, entry)
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)

    return allFiles

imagePaths = getListOfFiles("/content/drive/MyDrive/CSE - 340/animals/data") ## Folder structure: datasets --> sub-folders with labels name
print(imagePaths)
```

```
['/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00045.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00189.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00417.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00437.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00705.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00725.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00383.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00591.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00545.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00316.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00363.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00361.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00741.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00419.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00385.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00046.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00768.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00285.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00133.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00710.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00700.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00773.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00321.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00281.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00548.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00393.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00161.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00465.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00126.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00247.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00528.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00758.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00218.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00648.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00230.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00067.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00067.jpg']
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
yDrive/CSE - 340/animals/data/cats/cats_00986.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00973.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00961.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00965.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00997.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00994.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00971.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00995.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00969.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00968.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00972.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00987.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00991.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00985.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00983.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00975.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00966.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00984.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00981.jpg', '/content/drive/MyDrive/CSE - 340/animals/data/cats/cats_00964.jpg']
```

## To load the image dataset and separate as images and labels(target variable)

In [ ]:

```
sp = SimplePreprocessor(32, 32)
sdl = SimpleDatasetLoader(preprocessors=[sp])
(data, labels) = sdl.load(imagePaths, verbose=500)
data = data.reshape((data.shape[0], 3072))
# show some information on memory consumption of the images
print("[INFO] features matrix: {:.1f}MB".format(
    data.nbytes / (1024 * 1024.0)))

# encode the labels as integers
le = LabelEncoder()
labels = le.fit_transform(labels)
```

```
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] features matrix: 8.8MB
```

In [ ]:

```
##@title To load the dataset
sp = SimplePreprocessor(256, 256)
sdl = SimpleDatasetLoader(preprocessors=[sp])
(data1, labels1) = sdl.load(imagePaths, verbose=500)
# data = data.reshape((data.shape[0], 3072))
# show some information on memory consumption of the images
print("[INFO] features matrix: {:.1f}MB".format(
    data1.nbytes / (1024 * 1024.0)))
```

```
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] features matrix: 562.5MB
```

In [ ]:

```
##@title Label Encoding on target variable
le = LabelEncoder()
labels1 = le.fit_transform(labels1)
```

## Explore the data

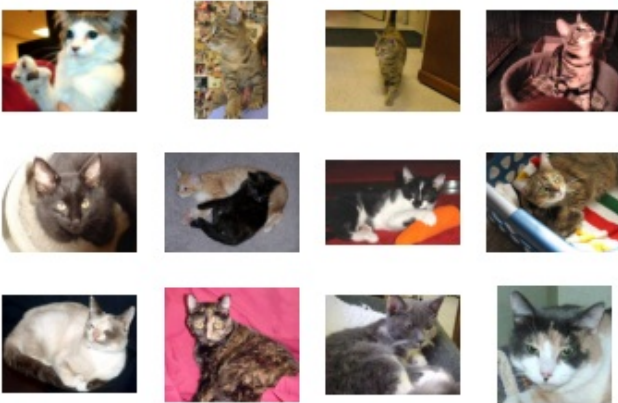
In [ ]:

```

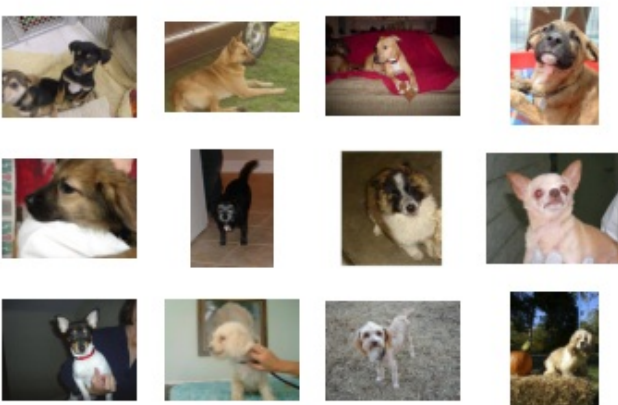
class_names = ['cat', 'dog', 'pandas']
import os
import matplotlib.pyplot as plt
path = '/content/drive/MyDrive/CSE - 340/animals/data/'
categories = ['cats', 'dogs', 'panda']
for category in categories:
    fig, _ = plt.subplots(3,4)
    fig.suptitle(category)
    fig.patch.set_facecolor('xkcd:white')
    for k, v in enumerate(os.listdir(path+category)[:12]):
        img = plt.imread(path+category+'/'+v)
        plt.subplot(3, 4, k+1)
        plt.axis('off')
        plt.imshow(img)
    plt.show()

```

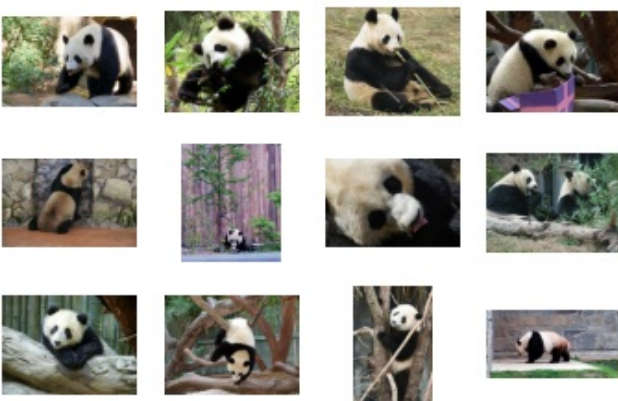
cats



dogs



panda



In [ ]:

```

trainX, testX, trainY, testY = train_test_split(data, labels,
        test_size=0.20, random_state=1)

```



## K - Nearest Neighbors - Base Learners

In [ ]:

```
knn_model = KNeighborsClassifier(n_neighbors=3)
knn_model.fit(trainX, trainY)
```

Out[ ]:

KNeighborsClassifier(n\_neighbors=3)

## METRICS AND MEASURE

In [ ]:

```
print(classification_report(testY, knn_model.predict(testX)))
```

	precision	recall	f1-score	support
0	0.42	0.64	0.50	208
1	0.40	0.45	0.43	186
2	0.92	0.33	0.48	206
accuracy			0.47	600
macro avg	0.58	0.47	0.47	600
weighted avg	0.58	0.47	0.47	600

In [ ]:

```
knn_y_pred = knn_model.predict(testX)
from sklearn.metrics import accuracy_score
k = accuracy_score(testY, knn_y_pred)
print("Accuracy: %.2f%%" % (k * 100.0))
```

Accuracy: 47.33%

## Decision Tree Classifier - Base Learner

In [ ]:

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(random_state=1, max_depth=None, criterion='gini')
dt_model.fit(trainX, trainY)
```

Out[ ]:

DecisionTreeClassifier(random\_state=1)

In [ ]:

```
dtmodel_pred = dt_model.predict(testX)
d = accuracy_score(testY, dtmodel_pred)
print("Accuracy is {}%".format(d*100))
```

Accuracy is 50.5%

In [ ]:

```
print(classification_report(testY, dt_model.predict(testX)))
```

	precision	recall	f1-score	support
0	0.45	0.44	0.44	208
1	0.41	0.43	0.42	186
2	0.66	0.64	0.65	206
accuracy			0.51	600

macro avg	0.50	0.50	0.50	600
weighted avg	0.51	0.51	0.51	600

## SVC - Base Learner

In [ ]:

```
from sklearn.svm import SVC
svc_model = SVC(random_state=1,probability=True)
svc_model.fit(trainX,trainY)
```

Out[ ]:

SVC(probability=True, random\_state=1)

In [ ]:

```
svcmodel_pred = svc_model.predict(testX)
a = accuracy_score(testY,svcmodel_pred)
print("Accuracy: %.2f%%" % (a * 100.0))
```

Accuracy: 62.67%

In [ ]:

```
print(classification_report(testY, svc_model.predict(testX)))
```

	precision	recall	f1-score	support
0	0.57	0.55	0.56	208
1	0.48	0.48	0.48	186
2	0.81	0.83	0.82	206
accuracy			0.63	600
macro avg	0.62	0.62	0.62	600
weighted avg	0.62	0.63	0.63	600

**MERITS AND DEMERITS: KNN(K nearest neighbor) gives an accuracy of 47.33% DecisionTreeClassifier gives 50.5% SVC gives 62.67% Therefore while looking at the individual models,SVC which falls under the category of Supervised learning algorithms gives the highest accuracy when compared to others as it uses the concept of Margin to classify between classes.**

## CO2: ENSEMBLE LEARNING

### ENSEMBLE DESIGN TECHNIQUES

## Voting Classifier

In [ ]:

```
estimators = [(['knn',knn_model)),(['decision_tree',dt_model)),(['svc',svc_model))]
```

Out[ ]:

```
[('knn', KNeighborsClassifier(n_neighbors=3)),
 ('decision_tree', DecisionTreeClassifier(random_state=1)),
 ('svc', SVC(probability=True, random_state=1))]
```

## Hard Voting

In [ ]:

```
from sklearn.ensemble import VotingClassifier
```

```
array([[1.33333333, 0.33333333, 1.66666667, 1.          , 0.66666667,
        0.33333333, 0.          , 0.66666667, 0.33333333, 0.33333333,
        1.          , 0.66666667, 0.          , 1.33333333, 0.33333333,
        2.          , 0.66666667, 0.33333333, 2.          , 2.          ,
        1.          , 0.33333333, 0.          , 1.66666667, 0.33333333,
        0.66666667, 0.33333333, 1.66666667, 0.66666667, 1.          ,
        1.33333333, 1.33333333, 0.          , 1.          , 0.33333333,
        1.66666667, 2.          , 0.66666667, 0.33333333, 1.33333333,
        0.33333333, 0.66666667, 1.33333333, 0.66666667, 0.66666667,
        0.33333333, 1.66666667, 1.66666667, 1.66666667, 0.33333333,
        2.          , 2.          , 0.33333333, 0.          , 0.66666667,
        0.          , 0.66666667, 1.33333333, 0.66666667, 0.66666667,
        1.33333333, 1.66666667, 1.          , 0.33333333, 0.66666667,
        0.66666667, 0.33333333, 1.          , 1.          , 0.33333333,
        1.          , 1.33333333, 1.33333333, 0.33333333, 0.33333333,
        1.          , 1.          , 1.33333333, 0.33333333, 1.66666667,
        0.          , 1.          , 0.33333333, 1.66666667, 0.33333333,
        0.33333333, 0.33333333, 1.          , 0.          , 2.          ,
        0.66666667, 0.66666667, 0.33333333, 1.66666667, 0.66666667,
        1.          , 1.          , 1.          , 0.66666667, 0.66666667,
        0.          , 1.33333333, 1.          , 0.33333333, 0.66666667,
```

0. , 1.33333333, 1. , 0.33333333, 0.66666667,  
1.66666667, 1. , 1.66666667, 0. , 0.66666667,  
1. , 0.66666667, 0.33333333, 0. , 2. ,  
1.33333333, 0.66666667, 2. , 0. , 0.66666667,  
0.33333333, 1. , 0.66666667, 0.33333333, 1.66666667,  
0. , 0.33333333, 1.33333333, 1.66666667, 1. ,  
2. , 1. , 0.66666667, 0.66666667, 0.66666667,  
0. , 1.66666667, 0.66666667, 0.33333333, 1. ,  
1. , 0.66666667, 0. , 1.33333333, 0. ,  
0.66666667, 0.66666667, 1.33333333, 0. , 1.33333333,  
0. , 1. , 0.33333333, 0.33333333, 0. ,  
0. , 1.33333333, 1.33333333, 1. , 0.66666667,  
1.66666667, 0. , 0. , 0.66666667, 0.33333333,  
0.66666667, 1.33333333, 0.66666667, 2. , 0.66666667,  
1.33333333, 0.33333333, 1.33333333, 0.66666667, 1. ,  
1.33333333, 2. , 0.33333333, 1.33333333, 0. ,  
1. , 0.66666667, 2. , 0.66666667, 0. ,  
0.66666667, 2. , 1.33333333, 0.66666667, 1.66666667,  
0.66666667, 0.66666667, 0. , 0.66666667, 0.33333333,  
0.66666667, 0.66666667, 0.66666667, 0.33333333, 0.66666667,  
0.33333333, 1.66666667, 0.33333333, 0.66666667, 2. ,  
0. , 0.33333333, 1. , 0. , 0.33333333,  
0.66666667, 0.66666667, 0.33333333, 0.33333333, 0. ,  
1.66666667, 1. , 1. , 0.66666667, 2. ,  
1.33333333, 1.33333333, 1. , 0.66666667, 1. ,  
0.66666667, 1. , 0.33333333, 1. , 0.33333333,  
0. , 1.66666667, 2. , 2. , 0.33333333,  
1.33333333, 1.33333333, 1. , 0.33333333, 1. ,  
1.33333333, 0.66666667, 0.66666667, 0. , 0.66666667,  
1.66666667, 1. , 0. , 0.66666667, 1. ,  
0.66666667, 0.33333333, 0.66666667, 0.33333333, 0. ,  
1. , 0. , 1. , 0.66666667, 0.33333333,  
1.33333333, 1.33333333, 1. , 0.66666667, 0.33333333,  
1. , 0.66666667, 0.33333333, 0.33333333, 0.66666667,  
1.33333333, 1.33333333, 0. , 1.33333333, 0. ,  
1.66666667, 1. , 1.66666667, 1.66666667, 2. ,  
0.33333333, 0.66666667, 0.66666667, 0. , 0.33333333,  
0.33333333, 2. , 0. , 2. , 2. ,  
0. , 2. , 2. , 1.33333333, 1.66666667,  
1.66666667, 1.66666667, 1. , 0.66666667, 0.66666667,  
0. , 1. , 0. , 1. , 0.66666667,  
2. , 0.66666667, 0. , 2. , 2. ,  
0.66666667, 0.33333333, 0.66666667, 1. , 0. ,  
0.66666667, 0.66666667, 0.66666667, 1.33333333, 0.33333333,  
0.33333333, 0.66666667, 0.66666667, 1.33333333, 0.33333333,  
0.33333333, 0. , 1.66666667, 0.33333333, 0.66666667,  
0.33333333, 1. , 0.66666667, 2. , 1. ,  
0.66666667, 0.33333333, 2. , 2. , 0.33333333,  
1. , 0. , 1.33333333, 0.66666667, 1. ,  
0.66666667, 1.33333333, 0.66666667, 1.66666667, 1.33333333,  
2. , 0. , 0.33333333, 2. , 0.33333333,  
0. , 0.66666667, 0.66666667, 0.66666667, 1. ,  
0.66666667, 1.33333333, 0.66666667, 1.33333333, 0.66666667,  
0.66666667, 1. , 0.33333333, 1.33333333, 1.33333333,  
0.66666667, 0.66666667, 0.66666667, 0.66666667, 0.66666667,  
0.66666667, 1.33333333, 0.33333333, 0.66666667, 1. ,  
1. , 1.33333333, 2. , 0.66666667, 0.33333333,  
1. , 0. , 0.66666667, 1.33333333, 0.33333333,  
1.66666667, 1.33333333, 2. , 1. , 0.66666667,  
1.33333333, 1. , 1.33333333, 0.66666667, 0.33333333,  
0.33333333, 0. , 2. , 1. , 1.66666667,  
1.33333333, 0. , 1. , 1. , 1. ,  
1.33333333, 2. , 0. , 1.33333333, 1. ,  
1.33333333, 0. , 1. , 0.66666667, 0.33333333,  
0.33333333, 0. , 0.33333333, 0.66666667, 1.33333333,  
0.66666667, 0. , 0.33333333, 0. , 1.33333333,  
0.66666667, 1.66666667, 0.66666667, 0. , 0.66666667,  
0.33333333, 0.33333333, 0.66666667, 1.33333333, 0. ,  
1.33333333, 1.33333333, 0. , 2. , 0.33333333,  
0.66666667, 0.33333333, 2. , 0. , 1. ,  
0. , 1.33333333, 0.33333333, 1.66666667, 2. ,  
0.33333333, 0.33333333, 0.66666667, 0.33333333, 1. ,  
n n 1 33333333 n 66666667 2

```

0. , 0. , 1.33333333, 0.66666667, 2. ,
1. , 1.33333333, 1.33333333, 2. , 1.33333333,
0.66666667, 1.33333333, 0.66666667, 1.33333333, 0.33333333,
0.66666667, 0. , 2. , 0. , 0.33333333,
0.66666667, 1.33333333, 0.33333333, 1.66666667, 0.66666667,
0.66666667, 1. , 0.33333333, 1.66666667, 0.33333333,
1. , 1. , 1. , 0.66666667, 1. ,
1. , 1. , 1.33333333, 2. , 2. ,
0. , 0. , 1. , 0.66666667, 0. ,
1.66666667, 0.66666667, 0.33333333, 0.33333333, 1.33333333,
0.66666667, 1.33333333, 1. , 0. , 1.33333333,
0.33333333, 1.66666667, 2. , 1.33333333, 0.66666667,
1.33333333, 0.66666667, 0.66666667, 0.66666667, 1. ,
2. , 1.33333333, 1. , 0.33333333, 0. ,
0.33333333, 1. , 2. , 0.66666667, 0.66666667,
0.66666667, 0.33333333, 0.66666667, 1.66666667, 1. ,
0.66666667, 0.66666667, 1. , 1.33333333, 1.33333333,
1. , 1. , 0.66666667, 1.33333333, 0.33333333,
0.66666667, 1.66666667, 0. , 0.66666667, 0.33333333,
0.33333333, 1.33333333, 1. , 2. , 1.33333333,
1.33333333, 2. , 2. , 1.33333333, 0.66666667,
1.33333333, 1. , 1. , 1. , 0.66666667,
0.66666667, 0.66666667, 0.66666667, 1.33333333, 0. ,
1.66666667, 1.33333333, 2. , 1. , 0.66666667,
1.33333333, 2. , 0. , 1. , 1.33333333,
0.66666667, 1. , 0.66666667, 1. , 2. ,
0.66666667, 1.33333333, 0.33333333, 0.66666667, 0.33333333,
0. , 1. , 0. , 1.33333333, 1.66666667])

```

## Weighted Averaging

In [ ]:

```

weighted_predictions=(knn_y_pred *0.3 + dtmodel_pred *0.4 + svcmodel_pred*0.3)
weighted_predictions

```

Out[ ]:

```

array([[1.4, 0.4, 1.6, 1. , 0.6, 0.4, 0. , 0.6, 0.4, 0.3, 0.9, 0.6, 0. ,
        1.3, 0.3, 2. , 0.7, 0.3, 2. , 2. , 1. , 0.3, 0. , 1.7, 0.4, 0.7,
        0.3, 1.7, 0.7, 1. , 1.4, 1.4, 0. , 1. , 0.3, 1.7, 2. , 0.6, 0.4,
        1.4, 0.3, 0.7, 1.4, 0.8, 0.7, 0.3, 1.7, 1.7, 1.7, 0.4, 2. , 2. ,
        0.4, 0. , 0.8, 0. , 0.7, 1.3, 0.6, 0.7, 1.3, 1.7, 1. , 0.4, 0.7,
        0.6, 0.4, 1. , 1. , 0.3, 1.1, 1.3, 1.2, 0.4, 0.3, 1. , 1. , 1.4,
        0.3, 1.6, 0. , 1. , 0.4, 1.7, 0.3, 0.3, 0.3, 1. , 0. , 2. , 0.7,
        0.6, 0.4, 1.7, 0.7, 1.1, 1.1, 1.1, 0.7, 0.6, 0. , 1.4, 1. , 0.3,
        0.6, 1.7, 1.1, 1.7, 0. , 0.8, 1. , 0.6, 0.4, 0. , 2. , 1.4, 0.6,
        2. , 0. , 0.6, 0.3, 1.1, 0.6, 0.3, 1.7, 0. , 0.3, 1.3, 1.7, 1. ,
        2. , 1.1, 0.7, 0.6, 0.7, 0. , 1.7, 0.6, 0.3, 1.1, 1. , 0.6, 0. ,
        1.4, 0. , 0.8, 0.7, 1.4, 0. , 1.3, 0. , 1. , 0.3, 0.4, 0. , 0. ,
        1.4, 1.3, 1. , 0.7, 1.7, 0. , 0. , 0.6, 0.3, 0.8, 1.4, 0.7, 2. ,
        0.6, 1.2, 0.4, 1.4, 0.6, 1. , 1.4, 2. , 0.4, 1.3, 0. , 1. , 0.7,
        2. , 0.7, 0. , 0.8, 2. , 1.4, 0.7, 1.7, 0.7, 0.6, 0. , 0.6, 0.3,
        0.7, 0.8, 0.8, 0.4, 0.7, 0.3, 1.7, 0.4, 0.7, 2. , 0. , 0.4, 1. ,
        0. , 0.3, 0.7, 0.7, 0.3, 0.3, 0. , 1.6, 1. , 1. , 0.7, 2. , 1.4,
        1.4, 1. , 0.6, 1. , 0.7, 1. , 0.3, 1. , 0.3, 0. , 1.7, 2. , 2. ,
        0.3, 1.4, 1.4, 1. , 0.4, 1. , 1.4, 0.6, 0.6, 0. , 0.7, 1.7, 1. ,
        0. , 0.7, 1. , 0.8, 0.3, 0.7, 0.4, 0. , 0.9, 0. , 0.9, 0.7, 0.3,
        1.4, 1.4, 1. , 0.6, 0.3, 1. , 0.7, 0.3, 0.4, 0.6, 1.4, 1.4, 0. ,
        1.4, 0. , 1.7, 1. , 1.7, 1.6, 2. , 0.3, 0.8, 0.7, 0. , 0.4, 0.3,
        2. , 0. , 2. , 2. , 0. , 2. , 2. , 1.4, 1.7, 1.7, 1.7, 1.1, 0.7,
        0.6, 0. , 1.1, 0. , 1. , 0.7, 2. , 0.8, 0. , 2. , 2. , 0.7, 0.4,
        0.6, 0.9, 0. , 0.6, 0.8, 0.6, 1.4, 0.4, 0.4, 0.6, 0.7, 1.4, 0.4,
        0.4, 0. , 1.7, 0.4, 0.7, 0.3, 1. , 0.8, 2. , 1.1, 0.7, 0.3, 2. ,
        2. , 0.3, 1. , 0. , 1.4, 0.6, 1. , 0.7, 1.4, 0.7, 1.7, 1.3, 2. ,
        0. , 0.3, 2. , 0.4, 0. , 0.7, 0.7, 0.7, 1. , 0.6, 1.4, 0.6, 1.4,
        0.7, 0.6, 1. , 0.4, 1.4, 1.3, 0.7, 0.7, 0.7, 0.7, 0.7, 0.6, 1.4,
        0.4, 0.7, 1. , 1. , 1.4, 2. , 0.7, 0.3, 1.1, 0. , 0.7, 1.4, 0.4,
        1.7, 1.4, 2. , 1. , 0.7, 1.4, 1. , 1.4, 0.7, 0.3, 0.3, 0. , 2. ,
        1. , 1.7, 1.4, 0. , 0.9, 0.9, 1. , 1.3, 2. , 0. , 1.4, 0.9, 1.4,
        0. , 1.1, 0.6, 0.4, 0.4, 0. , 0.3, 0.7, 1.4, 0.7, 0. , 0.3, 0. ,
        1.4, 0.7, 1.7, 0.6, 0. , 0.6, 0.3, 0.4, 0.6, 1.4, 0. , 1.4, 1.3

```

```
1.4, 0.7, 1.7, 0.0, 0. , 0.0, 0.3, 0.4, 0.0, 1.4, 0. , 1.4, 1.0,
0. , 2. , 0.4, 0.7, 0.4, 2. , 0. , 1.1, 0. , 1.2, 0.3, 1.7, 2. ,
0.4, 0.4, 0.7, 0.3, 1. , 0. , 0. , 1.4, 0.6, 2. , 1. , 1.4, 1.4,
2. , 1.2, 0.7, 1.4, 0.8, 1.4, 0.3, 0.6, 0. , 2. , 0. , 0.3, 0.8,
1.4, 0.4, 1.6, 0.7, 0.7, 0.9, 0.3, 1.7, 0.4, 1. , 1. , 1. , 0.7,
0.9, 1. , 1. , 1.4, 2. , 2. , 0. , 0. , 1. , 0.7, 0. , 1.7, 0.7,
0.3, 0.4, 1.4, 0.7, 1.4, 1. , 0. , 1.2, 0.3, 1.7, 2. , 1.4, 0.6,
1.4, 0.6, 0.6, 0.6, 1. , 2. , 1.2, 1. , 0.4, 0. , 0.3, 1.1, 2. ,
0.6, 0.6, 0.7, 0.3, 0.6, 1.6, 1.1, 0.6, 0.7, 0.9, 1.4, 1.4, 1.1,
1. , 0.6, 1.4, 0.3, 0.7, 1.7, 0. , 0.7, 0.3, 0.3, 1.2, 1.1, 2. ,
1.4, 1.4, 2. , 2. , 1.3, 0.6, 1.4, 1.1, 1.1, 1.1, 0.7, 0.6, 0.6,
0.7, 1.4, 0. , 1.7, 1.4, 2. , 1.1, 0.8, 1.4, 2. , 0. , 1.1, 1.4,
0.7, 1.1, 0.6, 1.1, 2. , 0.8, 1.4, 0.3, 0.6, 0.4, 0. , 1. , 0. ,
1.4, 1.7])
```

## Bagging Classifier

In [ ]:

```
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(base_estimator=SVC(),
                             random_state=1, n_estimators=10)
# print(bag_clf)
bag_clf.fit(trainX, trainY)
```

Out[ ]:

```
BaggingClassifier(base_estimator=SVC(), random_state=1)
```

In [ ]:

```
bag_pred = bag_clf.predict(testX)
b = accuracy_score(testY, bag_pred)
print("Accuracy is {}".format(b*100))
```

Accuracy is 63.0%

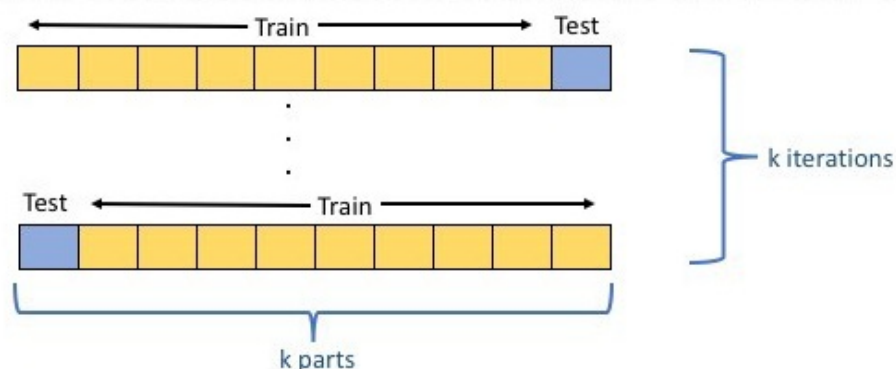
## ALGORITHM

### K-Fold

#### K-Fold - Algorithm

## K Folds Cross Validation Method

1. Divide the sample data into k parts.
2. Use k-1 of the parts for training, and 1 for testing.
3. Repeat the procedure k times, rotating the test set.
4. Determine an expected performance metric (mean square error, misclassification error rate, confidence interval, or other appropriate metric) based on the results across the iterations



In [ ]:

```
import numpy as np
from sklearn.model_selection import KFold

kf = KFold(n_splits=3)
kf.get_n_splits(trainX)
print(kf)
for train_index, test_index in kf.split(trainX):
    print("TRAIN:", train_index.shape, "TEST:", test_index.shape)
    kX_train, kX_test = trainX[train_index], trainX[test_index]
    ky_train, ky_test = trainY[train_index], trainY[test_index]
```

```
KFold(n_splits=3, random_state=None, shuffle=False)
TRAIN: (1600,) TEST: (800,)
TRAIN: (1600,) TEST: (800,)
TRAIN: (1600,) TEST: (800,)
```

In [ ]:

```
from sklearn.svm import SVC
svc2_model = SVC()
svc2_model.fit(kX_train, ky_train)
svcmodel_pred1 = svc2_model.predict(testX)
s = accuracy_score(testY, svcmodel_pred1)
print("Accuracy: %.2f%%" % (s * 100.0))
```

Accuracy: 62.17%

## Gradient Boosting

In [ ]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score
clf_GD = GradientBoostingClassifier(n_estimators=10, learning_rate=1.0, max_depth=1, random_state=0)
# print(clf_GD)
clf_GD.fit(trainX, trainY)
```

Out[ ]:

```
GradientBoostingClassifier(learning_rate=1.0, max_depth=1, n_estimators=10,
                           random_state=0)
```

In [ ]:

```
scores = cross_val_score(clf_GD, trainX, trainY, cv=5)
print(scores.mean())
```

0.5379166666666666

In [ ]:

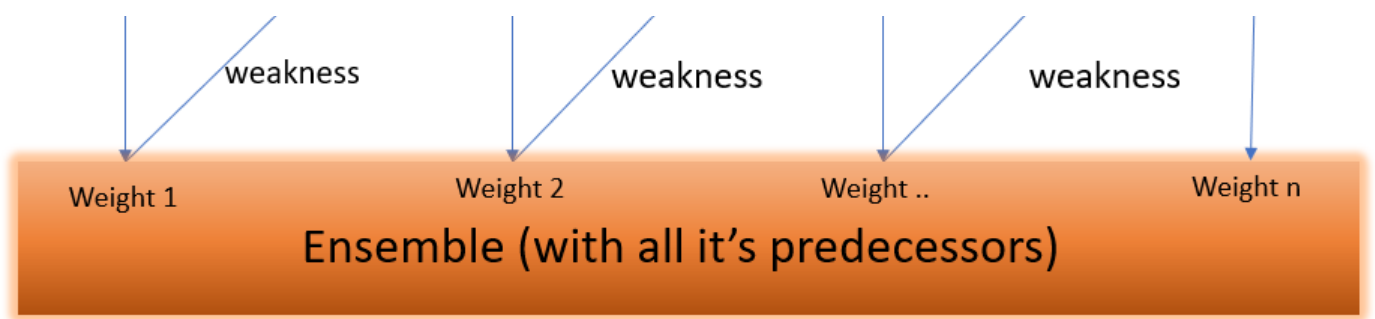
```
GD_pred = clf_GD.predict(testX)
GD = accuracy_score(testY, GD_pred)
print("Accuracy: %.2f%%" % (GD * 100.0))
```

Accuracy: 58.17%

## Ada Boost Classification







In [ ]:

```
from sklearn.ensemble import AdaBoostClassifier
clf_AB = AdaBoostClassifier(n_estimators=10)
scores = cross_val_score(clf_AB, trainX ,trainY, cv=5)
print(scores.mean())
```

0.5183333333333333

In [ ]:

```
clf_AB.fit(trainX ,trainY)
AB_pred = clf_AB.predict(testX)
AB = accuracy_score(testY,AB_pred)
print("Accuracy: %.2f%%" % (AB * 100.0))
```

Accuracy: 57.00%

## Tree Based Learning Methods



In [ ]:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(random_state = 0, n_estimators = 100,\
                                   criterion = 'entropy', max_leaf_nodes= 20,oob_score
= True, n_jobs = -1 )
# fit the model
classifier.fit(trainX, trainY)
```

Out[ ]:

```
RandomForestClassifier(criterion='entropy', max_leaf_nodes=20, n_jobs=-1,
                        oob_score=True, random_state=0)
```

In [ ]:

```
rf_pred = classifier.predict(testX)
rf = accuracy_score(testY,rf_pred)
print("Accuracy: %.2f%%" % (rf * 100.0))
```

Accuracy: 61.50%

In [ ]:

```
print(classification_report(testY, classifier.predict(testX),target_names=class_names))
```

	precision	recall	f1-score	support
cat	0.67	0.36	0.47	208
dog	0.48	0.64	0.55	186
pandas	0.72	0.85	0.78	206
accuracy			0.61	600
macro avg	0.63	0.62	0.60	600
weighted avg	0.63	0.61	0.60	600

# CO3: FEATURE SELECTION

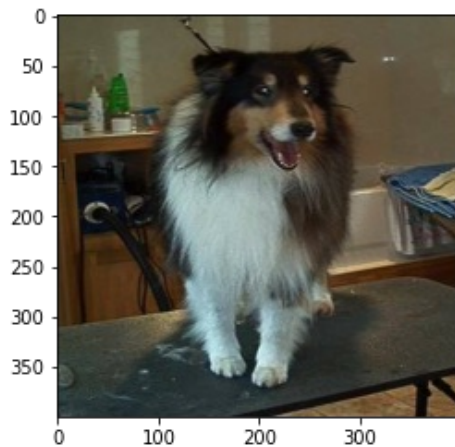
## DIMENSIONALITY REDUCTION- Principal Component Analysis

In [ ]:

```
import cv2
img = cv2.cvtColor(cv2.imread('/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_0007.jpg'), cv2.COLOR_BGR2RGB)
```

In [ ]:

```
import matplotlib.pyplot as plt
img = cv2.resize(img, (400, 400))
plt.imshow(img)
plt.show()
```

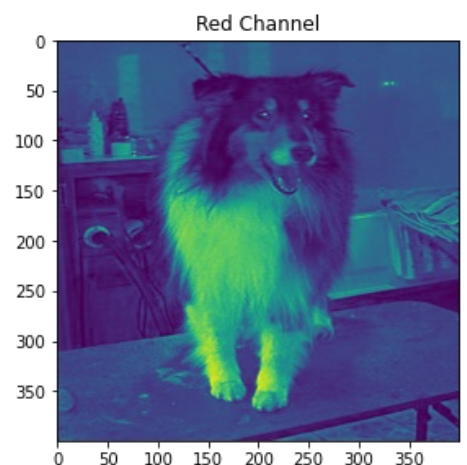
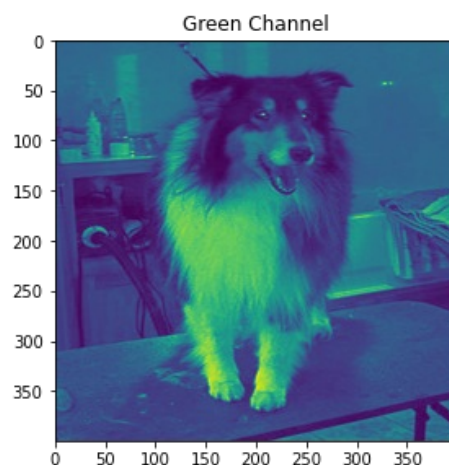
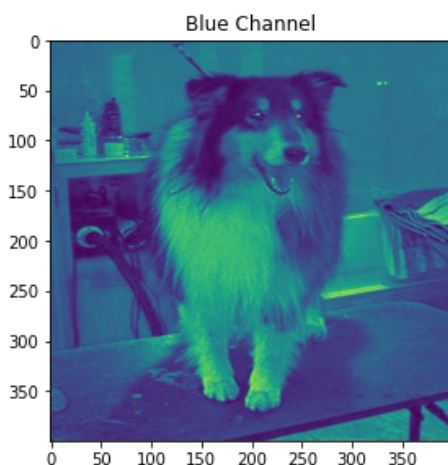


## FEATURE SELECTION AND EXTRACTION

**Now, We'll split the image into 3 channels and display each image:**

In [ ]:

```
#Splitting into channels
blue, green, red = cv2.split(img)
# Plotting the images
fig = plt.figure(figsize = (15, 7.2))
fig.add_subplot(131)
plt.title("Blue Channel")
plt.imshow(blue)
fig.add_subplot(132)
plt.title("Green Channel")
plt.imshow(green)
fig.add_subplot(133)
plt.title("Red Channel")
plt.imshow(red)
plt.show()
```



In [ ]:

```
import pandas as pd
blue_temp_df = pd.DataFrame(data = blue)
blue_temp_df
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	82	82	82	82	82	82	82	82	82	82	79	80	81	82	83	83	82	81	80	79	83	84	84	84	84	85	85	86	86	86	85
1	84	84	84	84	84	84	84	84	84	84	81	82	83	83	84	84	83	83	82	81	84	84	84	84	84	85	85	86	86	86	86
2	86	86	86	86	86	86	86	86	86	86	83	84	85	85	86	86	85	85	84	83	84	84	84	84	85	85	86	86	86	86	88
3	87	87	87	87	87	87	87	87	87	87	84	85	85	85	85	85	85	85	85	84	84	84	84	85	85	86	86	87	87	87	90
4	85	85	85	85	85	85	85	85	85	85	85	84	84	84	84	84	84	84	84	85	85	85	85	85	85	86	86	87	87	87	90
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	28	46	59	61	52	45	48	52	51	46	46	50	54	57	55	52	49	51	57	62	54	54	53	51	47	43	43	46	51	54	46
396	37	48	54	51	43	40	48	56	58	56	49	54	61	64	60	54	50	51	53	56	46	48	50	51	50	47	47	49	51	53	43
397	11	24	39	52	61	66	70	70	67	63	60	67	72	72	65	56	51	49	49	48	36	41	48	54	59	59	55	51	47	43	38
398	0	11	33	58	80	88	83	76	71	70	66	72	74	70	61	53	51	50	48	46	36	39	44	49	54	56	53	48	41	37	38
399	31	35	45	59	73	74	67	67	73	82	66	68	68	62	53	48	49	51	51	49	43	41	39	38	39	43	44	42	39	37	41

400 rows x 400 columns



Now we will divide all the data of all channels by 255 so that the data is scaled between 0 and 1.

In [ ]:

```
df_blue = blue/255
df_green = green/255
df_red = red/255
df_blue.shape
```

Out[ ]:

(400, 400)

## Fit and transform the data in PCA

We already have seen that each channel has 400 dimensions, and we will now consider only 50 dimensions for PCA and fit and transform the data and check how much variance is explained after reducing data to 50 dimensions.

In [ ]:

```
from sklearn.decomposition import PCA
import cv2
from scipy.stats import stats
import matplotlib.image as mpimg
pca_b = PCA(n_components=50)
pca_b.fit(df_blue)
trans_pca_b = pca_b.transform(df_blue)
pca_g = PCA(n_components=50)
pca_g.fit(df_green)
trans_pca_g = pca_g.transform(df_green)
pca_r = PCA(n_components=50)
pca_r.fit(df_red)
trans_pca_r = pca_r.transform(df_red)
```

In [ ]:

```
print(trans_pca_b.shape)
print(trans_pca_r.shape)
print(trans_pca_g.shape)
```

```
(400, 50)
(400, 50)
(400, 50)
```

**That is as expected. Let's check the sum of explained variance ratios of the 50 PCA components (i.e. most dominated 50 Eigenvalues) for each channel.**

In [ ]:

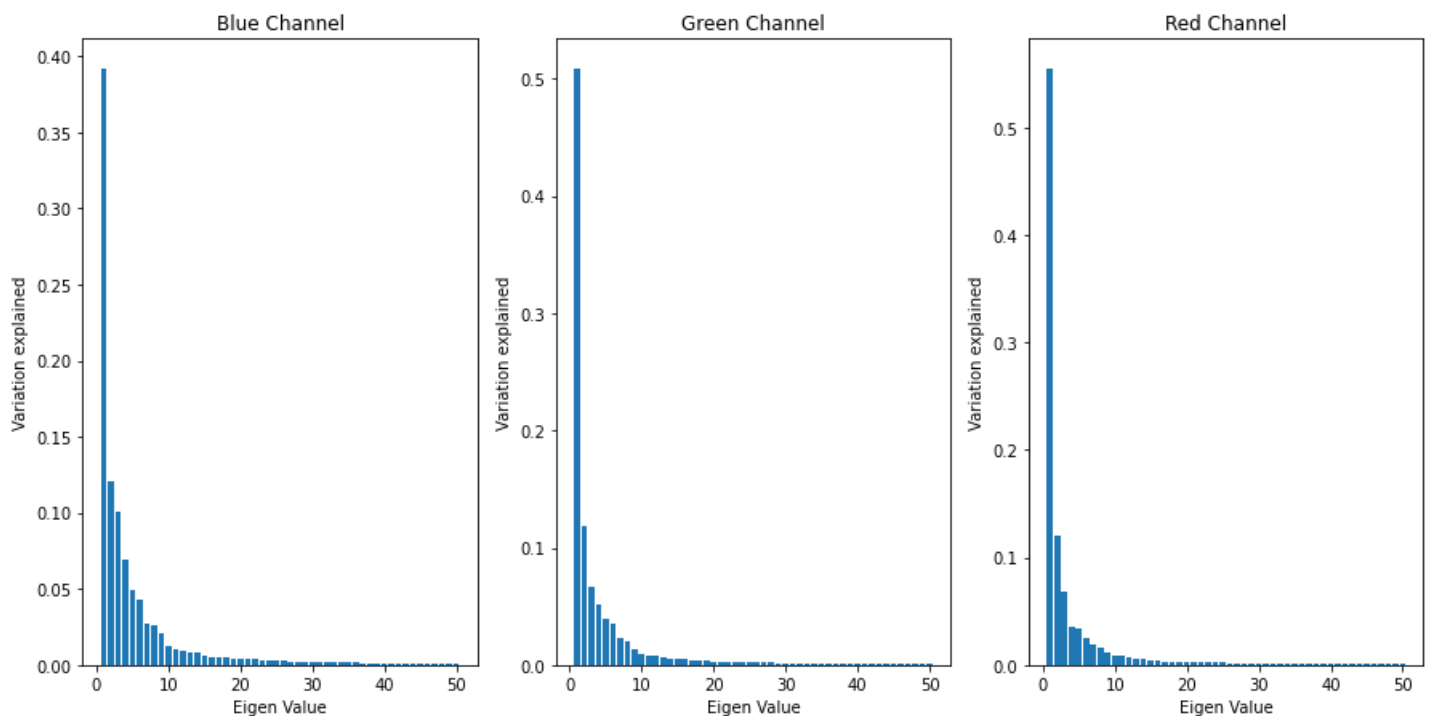
```
print(f"Blue Channel : {sum(pca_b.explained_variance_ratio_)}")
print(f"Green Channel: {sum(pca_g.explained_variance_ratio_)}")
print(f"Red Channel  : {sum(pca_r.explained_variance_ratio_)}")
```

```
Blue Channel : 0.9767877202984008
Green Channel: 0.9802287285800566
Red Channel  : 0.980097084816899
```

**Let's plot bar charts to check the explained variance ratio by each Eigenvalues separately for each of the 3 channels:**

In [ ]:

```
fig = plt.figure(figsize = (15, 7.2))
fig.add_subplot(131)
plt.title("Blue Channel")
plt.ylabel('Variation explained')
plt.xlabel('Eigen Value')
plt.bar(list(range(1,51)),pca_b.explained_variance_ratio_)
fig.add_subplot(132)
plt.title("Green Channel")
plt.ylabel('Variation explained')
plt.xlabel('Eigen Value')
plt.bar(list(range(1,51)),pca_g.explained_variance_ratio_)
fig.add_subplot(133)
plt.title("Red Channel")
plt.ylabel('Variation explained')
plt.xlabel('Eigen Value')
plt.bar(list(range(1,51)),pca_r.explained_variance_ratio_)
plt.show()
```



**Reconstruct the image and visualize**

## Reconstruct the image and visualize

We have completed our PCA dimensionality reduction. Now we will visualize the image again and for that, we have to reverse transform the data first and then merge the data of all the 3 channels into one. Let's proceed.

In [ ]:

```
b_arr = pca_b.inverse_transform(trans_pca_b)
g_arr = pca_g.inverse_transform(trans_pca_g)
r_arr = pca_r.inverse_transform(trans_pca_r)
print(b_arr.shape, g_arr.shape, r_arr.shape)
```

```
(400, 400) (400, 400) (400, 400)
```

In [ ]:

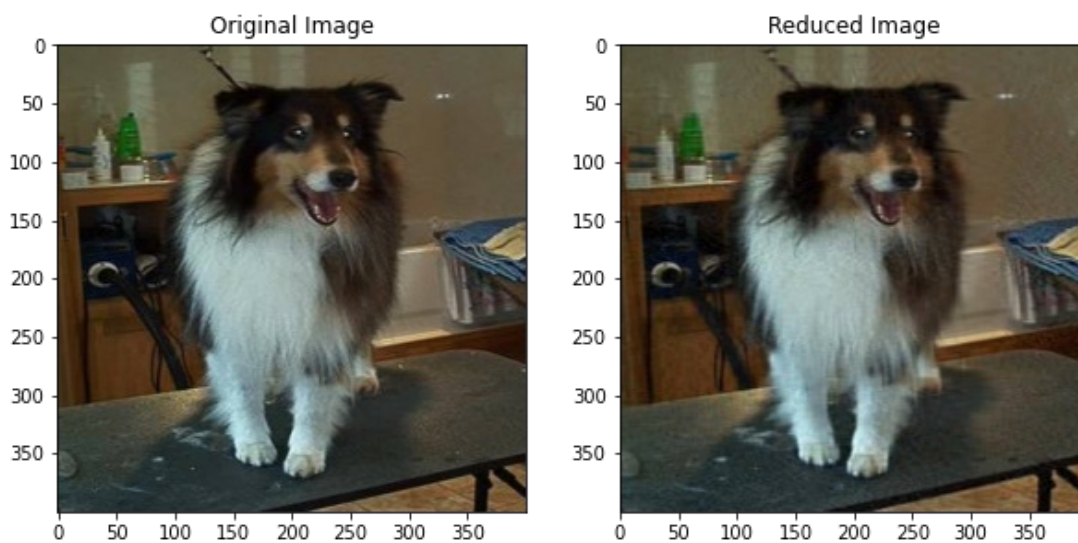
```
img_reduced= (cv2.merge((b_arr, g_arr, r_arr)))
print(img_reduced.shape)
```

```
(400, 400, 3)
```

In [ ]:

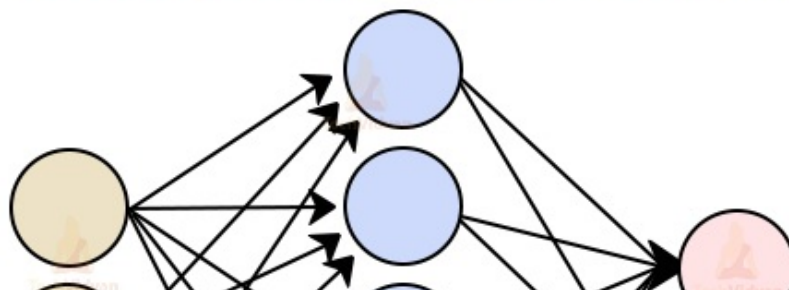
```
fig = plt.figure(figsize = (10, 7.2))
fig.add_subplot(121)
plt.title("Original Image")
plt.imshow(img)
fig.add_subplot(122)
plt.title("Reduced Image")
plt.imshow(img_reduced)
plt.show()
```

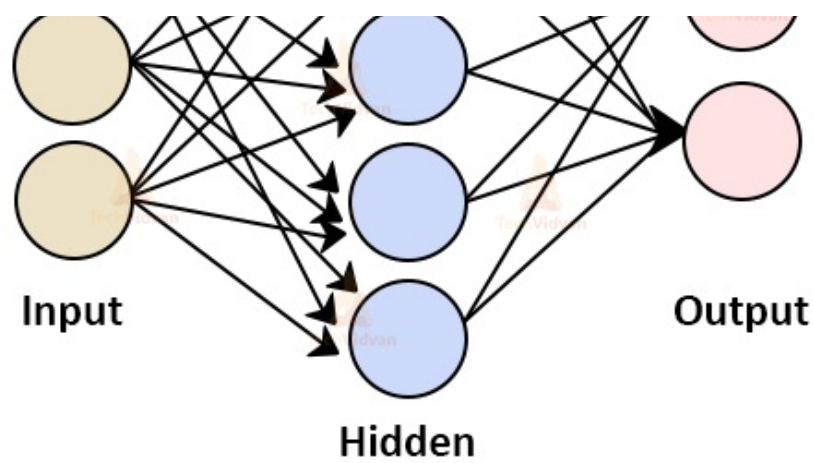
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



## CO4: Building Artificial Neural Network

### Architecture of Artificial Neural Network



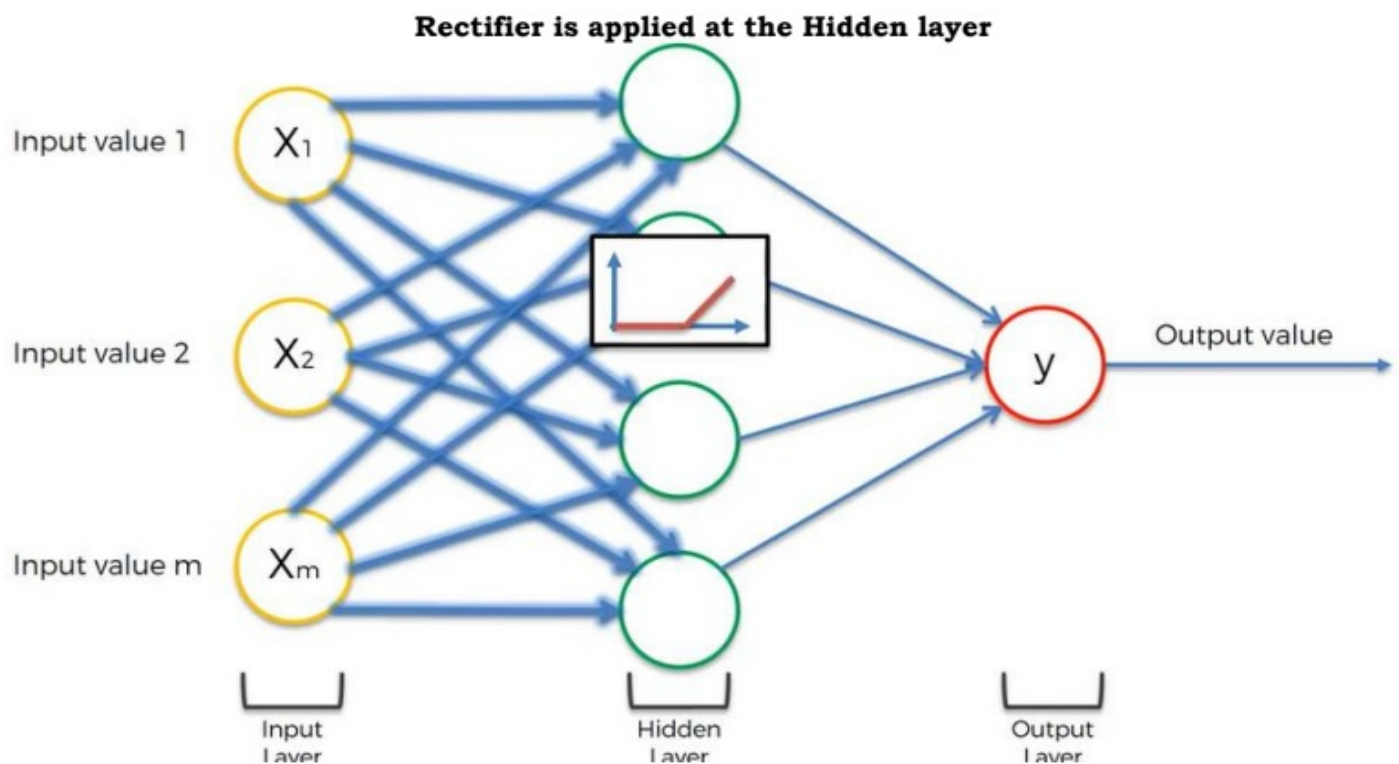


## Splitting the train and test data

In [ ]:

```
Xtrain,Xtest,Ytrain,Ytest = train_test_split(data1,labels1,test_size=0.2,random_state=0)
```

## Building the model



## DEVELOPMENT OF ANN

In [ ]:

```
ann_model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(256,256,3)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(3),
])
```

## Compling the ANN model

In [ ]:

```
In [ ]:
ann_model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                  metrics=['accuracy'])
```

## Fit the model to the train data

In [ ]:

```
ann_model.fit(Xtrain,Ytrain,epochs = 30,batch_size=16)
```

```
Epoch 1/30
150/150 [=====] - 19s 123ms/step - loss: 5.9669 - accuracy: 0.34
54
Epoch 2/30
150/150 [=====] - 18s 122ms/step - loss: 5.4117 - accuracy: 0.32
50
Epoch 3/30
150/150 [=====] - 19s 123ms/step - loss: 5.4120 - accuracy: 0.34
33
Epoch 4/30
150/150 [=====] - 19s 126ms/step - loss: 9.2797 - accuracy: 0.32
96
Epoch 5/30
150/150 [=====] - 18s 123ms/step - loss: 10.1471 - accuracy: 0.3
388
Epoch 6/30
150/150 [=====] - 18s 122ms/step - loss: 4.8903 - accuracy: 0.33
88
Epoch 7/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 8/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 9/30
150/150 [=====] - 19s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 10/30
150/150 [=====] - 19s 124ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 11/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 12/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 13/30
150/150 [=====] - 19s 125ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 14/30
150/150 [=====] - 19s 124ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 15/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 16/30
150/150 [=====] - 18s 121ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 17/30
150/150 [=====] - 19s 126ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 18/30
150/150 [=====] - 19s 127ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 19/30
150/150 [=====] - 19s 124ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 20/30
150/150 [=====] - 19s 123ms/step - loss: 1.0986 - accuracy: 0.33
oo
```



```

00
Epoch 21/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 22/30
150/150 [=====] - 19s 124ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 23/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 24/30
150/150 [=====] - 18s 123ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 25/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 26/30
150/150 [=====] - 19s 124ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 27/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 28/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 29/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88
Epoch 30/30
150/150 [=====] - 18s 122ms/step - loss: 1.0986 - accuracy: 0.33
88

```

Out[ ]:

<keras.callbacks.History at 0x7f4de00cf710>

In [ ]:

```

#@title Function to plot predictions on test images
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})"
               .format(class_names[predicted_label],
                       100*np.max(predictions_array),
                       class_names[true_label]),
               color=color)

def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')

```

## Prediction on a single image

In [ ]:

```
from keras.preprocessing import image
test_image=image.load_img(r'/content/drive/MyDrive/CSE - 340/animals/data/dogs/dogs_00085.jpg',
                           target_size=(256,256,3))
test_image=image.img_to_array(test_image)
test_image=np.expand_dims(test_image,axis=0)
print("Predicted Class :",np.argmax(ann_model.predict(test_image)))
```

Predicted Class : 2

## To make predictions

In [ ]:

```
probability_model = tf.keras.Sequential([ann_model,
                                         tf.keras.layers.Softmax()])
```

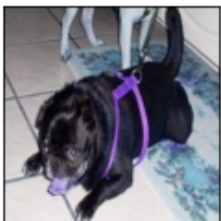
In [ ]:

```
predictions = probability_model.predict(Xtest)
```

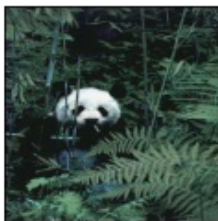
## Predictions on multiple test images

In [ ]:

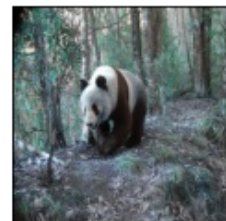
```
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], Ytest, Xtest)
    # plt.subplot(num_rows, 2*num_cols, 2*i+2)
    # plot_value_array(i, predictions[i], Ytest)
plt.tight_layout()
plt.show()
```



pandas 57% (dog)



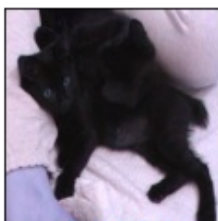
pandas 57% (pandas)



pandas 57% (pandas)



pandas 57% (cat)



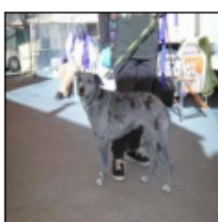
pandas 57% (cat)



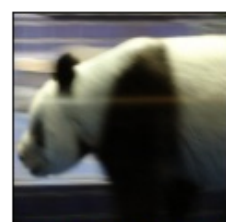
pandas 51% (cat)



pandas 57% (cat)



pandas 57% (dog)



pandas 57% (pandas)

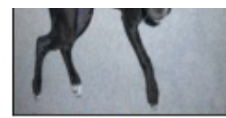




pandas 57% (cat)



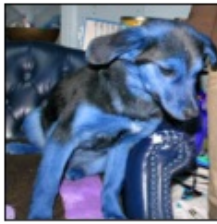
pandas 57% (pandas)



pandas 57% (dog)



pandas 57% (dog)



pandas 57% (dog)



pandas 57% (pandas)

## Preceptron Model - With 100 iterations

In [ ]:

```

from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import Perceptron
# define model
model2 = Perceptron(eta0=0.0001)
# define model evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid
grid = dict()
grid['max_iter'] = [1, 10, 50, 100]
# define search
search = GridSearchCV(model2, grid, scoring='accuracy', cv=cv, n_jobs=-1)
# perform the search
results = search.fit(trainX, trainY)
# summarize
print('Mean Accuracy: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)
# summarize all
means = results.cv_results_['mean_test_score']
params = results.cv_results_['params']
for mean, param in zip(means, params):
    print(">%.3f with: %r" % (mean, param))

```

```

Mean Accuracy: 0.521
Config: {'max_iter': 100}
>0.502 with: {'max_iter': 1}
>0.513 with: {'max_iter': 10}
>0.516 with: {'max_iter': 50}
>0.521 with: {'max_iter': 100}

```

## Homogenous Ensemble

In [ ]:

```

#@title Importing libraries
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras import optimizers
from sklearn.metrics import mean_squared_error

```

In [ ]:

```

import pandas as pd
from sklearn.utils import resample
from sklearn.metrics import precision_score, recall_score
accuracy = pd.DataFrame( columns=["Accuracy", "Precision", "Recall"])
predictions = np.zeros(shape=(10000, 7))

```

```

row_index = 0
for i in range(7):
    # bootstrap sampling
    boot_train = resample(Xtrain,Ytrain,replace=True, n_samples=20, random_state=None)

    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(256, 256, 3)),
        tf.keras.layers.Dense(256, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

    # compile the model
    model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

    # Train the model
    model.fit(Xtrain,Ytrain,epochs=5,batch_size=64)

    # Evaluate accuracy
    score = model.evaluate(Xtest, Ytest, batch_size=64)
    accuracy.loc[row_index, "Accuracy"] = score[1]

    # Make predictions
    model_pred= model.predict(Xtest)
    pred_classes =model_pred.argmax(axis=-1)
    accuracy.loc[row_index, 'Precision'] = precision_score(Ytest, pred_classes, average='weighted')
    accuracy.loc[row_index, 'Recall'] = recall_score(Ytest, pred_classes,average='weighted')

    # Save predictions to predictions array
    # predictions[:,i] = pred_classes

    print(score)
    row_index+=1

    print("Iteration " + str(i+1)+ " Accuracy : " + "{0}".format(score[1]))

```

```

Epoch 1/5
38/38 [=====] - 17s 405ms/step - loss: 35.4818 - accuracy: 0.3338
Epoch 2/5
38/38 [=====] - 16s 414ms/step - loss: 2.3912 - accuracy: 0.3308
Epoch 3/5
38/38 [=====] - 16s 415ms/step - loss: 1.2001 - accuracy: 0.4263
Epoch 4/5
38/38 [=====] - 15s 406ms/step - loss: 1.0252 - accuracy: 0.4708
Epoch 5/5
38/38 [=====] - 15s 404ms/step - loss: 0.9884 - accuracy: 0.4950
10/10 [=====] - 2s 141ms/step - loss: 0.9293 - accuracy: 0.5000
[0.9292542338371277, 0.5]
Iteration 1 Accuracy : 0.5
Epoch 1/5
38/38 [=====] - 17s 409ms/step - loss: 58.5122 - accuracy: 0.3454
Epoch 2/5
38/38 [=====] - 16s 410ms/step - loss: 2.8624 - accuracy: 0.3363

```

Epoch 3/5  
38/38 [=====] - 16s 410ms/step - loss: 1.0199 - accuracy: 0.4988  
Epoch 4/5  
38/38 [=====] - 16s 408ms/step - loss: 0.9753 - accuracy: 0.5275  
Epoch 5/5  
38/38 [=====] - 16s 411ms/step - loss: 0.8968 - accuracy: 0.5487  
10/10 [=====] - 2s 142ms/step - loss: 0.8847 - accuracy: 0.5317  
[0.8846628069877625, 0.5316666960716248]  
Iteration 2 Accuracy : 0.5316666960716248  
Epoch 1/5  
38/38 [=====] - 17s 416ms/step - loss: 27.6828 - accuracy: 0.3158  
Epoch 2/5  
38/38 [=====] - 16s 411ms/step - loss: 11.8826 - accuracy: 0.3383  
Epoch 3/5  
38/38 [=====] - 16s 421ms/step - loss: 2.3907 - accuracy: 0.3500  
Epoch 4/5  
38/38 [=====] - 16s 410ms/step - loss: 1.2331 - accuracy: 0.3633  
Epoch 5/5  
38/38 [=====] - 16s 410ms/step - loss: 1.0485 - accuracy: 0.4617  
10/10 [=====] - 2s 141ms/step - loss: 1.0340 - accuracy: 0.4983  
[1.034006953239441, 0.4983333349227905]  
Iteration 3 Accuracy : 0.4983333349227905  
Epoch 1/5  
38/38 [=====] - 17s 411ms/step - loss: 36.1788 - accuracy: 0.3113  
Epoch 2/5  
38/38 [=====] - 16s 420ms/step - loss: 6.4546 - accuracy: 0.3417  
Epoch 3/5  
38/38 [=====] - 16s 413ms/step - loss: 1.1705 - accuracy: 0.4338  
Epoch 4/5  
38/38 [=====] - 16s 420ms/step - loss: 0.9258 - accuracy: 0.5275  
Epoch 5/5  
38/38 [=====] - 16s 424ms/step - loss: 0.8872 - accuracy: 0.5342  
10/10 [=====] - 2s 142ms/step - loss: 0.9319 - accuracy: 0.4550  
[0.9318941235542297, 0.45500001311302185]  
Iteration 4 Accuracy : 0.45500001311302185  
Epoch 1/5  
38/38 [=====] - 17s 412ms/step - loss: 33.5043 - accuracy: 0.3313  
Epoch 2/5  
38/38 [=====] - 16s 412ms/step - loss: 5.1959 - accuracy: 0.3579  
Epoch 3/5  
38/38 [=====] - 16s 415ms/step - loss: 1.0092 - accuracy: 0.4771  
Epoch 4/5  
38/38 [=====] - 16s 417ms/step - loss: 0.9521 - accuracy: 0.5200  
Epoch 5/5  
38/38 [=====] - 16s 417ms/step - loss: 0.8876 - accuracy: 0.5425  
10/10 [=====] - 2s 144ms/step - loss: 0.9181 - accuracy: 0.5067  
[0.9181051850318909, 0.5066666603088379]  
Iteration 5 Accuracy : 0.5066666603088379  
Epoch 1/5  
38/38 [=====] - 17s 408ms/step - loss: 38.2352 - accuracy: 0.3442  
Epoch 2/5  
38/38 [=====] - 16s 412ms/step - loss: 4.8470 - accuracy: 0.3479  
Epoch 3/5  
38/38 [=====] - 16s 410ms/step - loss: 2.9944 - accuracy: 0.3625  
Epoch 4/5  
38/38 [=====] - 16s 409ms/step - loss: 0.9912 - accuracy: 0.4804  
Epoch 5/5  
38/38 [=====] - 16s 413ms/step - loss: 0.9270 - accuracy: 0.5292  
10/10 [=====] - 2s 151ms/step - loss: 1.0656 - accuracy: 0.4067  
[1.0656355619430542, 0.40666666626930237]  
Iteration 6 Accuracy : 0.40666666626930237  
Epoch 1/5  
38/38 [=====] - 21s 503ms/step - loss: 35.7979 - accuracy: 0.3187  
Epoch 2/5  
38/38 [=====] - 16s 408ms/step - loss: 3.9657 - accuracy: 0.3521  
Epoch 3/5

```
38/38 [=====] - 16s 409ms/step - loss: 1.2895 - accuracy: 0.4313
Epoch 4/5
38/38 [=====] - 16s 415ms/step - loss: 0.9892 - accuracy: 0.5029
Epoch 5/5
38/38 [=====] - 15s 407ms/step - loss: 0.9021 - accuracy: 0.5387
10/10 [=====] - 2s 141ms/step - loss: 1.2063 - accuracy: 0.4950
[1.2063485383987427, 0.4950000047683716]
Iteration 7 Accuracy : 0.4950000047683716
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
_warn_prf(average, modifier, msg_start, len(result))
```

## ACCURACY MEASURE AND ASSESSMENT

### Predictions on multiple test images

In [ ]:

```
num_rows = 7
num_cols = 4
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, model_pred[i], Ytest, Xtest)
    # plt.subplot(num_rows, 2*num_cols, 2*i+2)
    # plot_value_array(i, predictions[i], Ytest)
plt.tight_layout()
plt.show()
```



dog 56% (dog)



dog 49% (pandas)



pandas 81% (pandas)



dog 70% (cat)



dog 78% (cat)



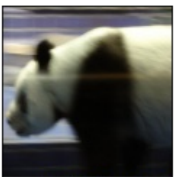
dog 58% (cat)



pandas 89% (cat)



pandas 53% (dog)



pandas 93% (pandas)



pandas 90% (cat)



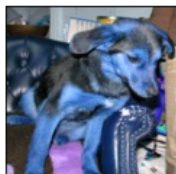
pandas 98% (pandas)



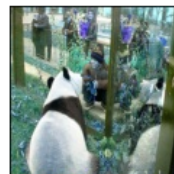
dog 81% (dog)



pandas 76% (dog)



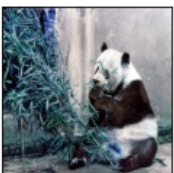
dog 62% (dog)



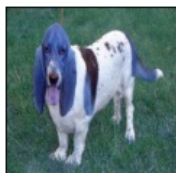
pandas 53% (pandas)



dog 50% (cat)



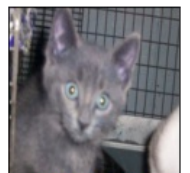
pandas 99% (pandas)



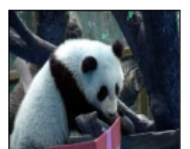
pandas 90% (dog)



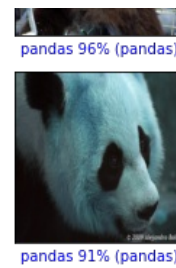
pandas 48% (dog)



dog 53% (cat)

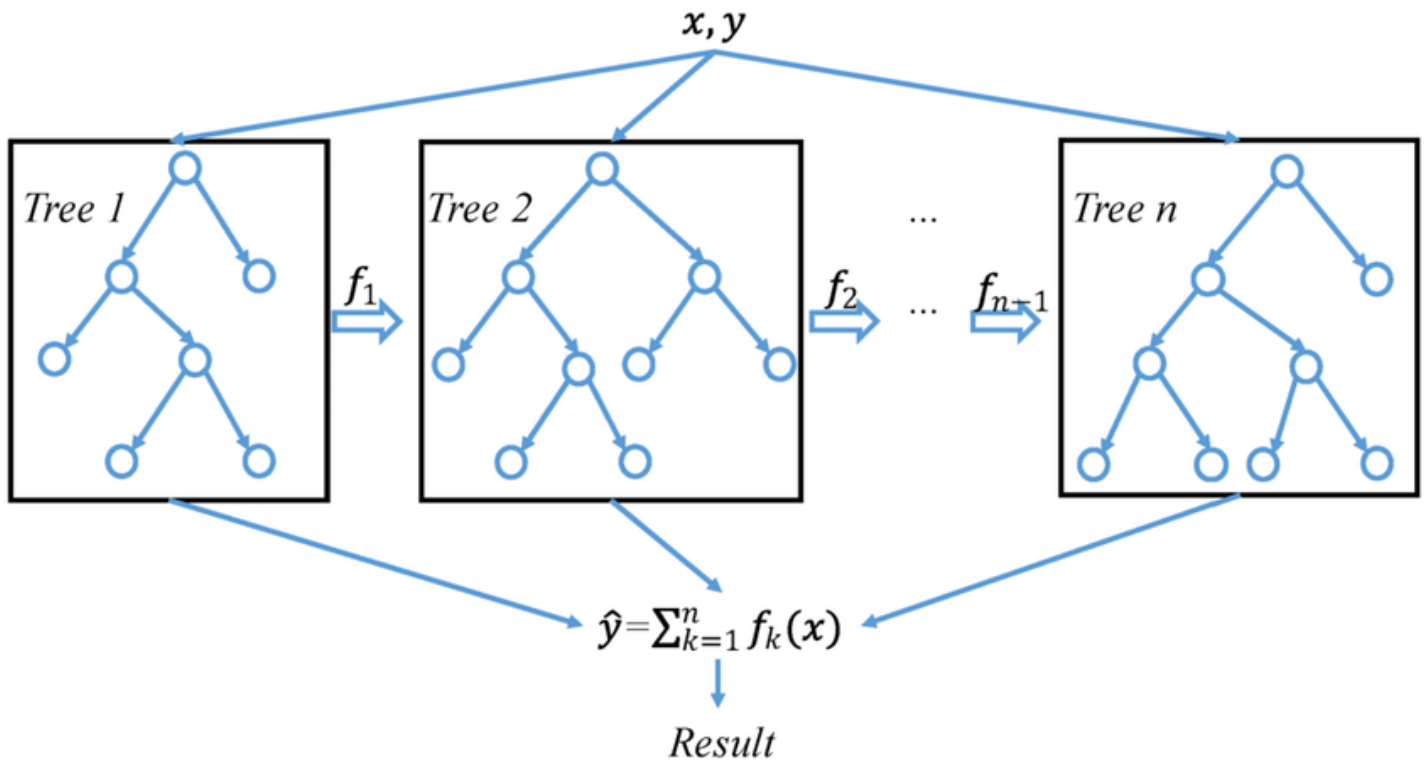






## XGBoost

### XGBoost - Architecture



In [ ]:

```
import warnings
warnings.filterwarnings('ignore')
import os
import pandas as pd
import numpy as np
from numpy import sort
from xgboost import XGBClassifier
from xgboost import plot_tree
from xgboost import plot_importance
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split, KFold, cross_val_score, StratifiedKFold
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix
import itertools
```

In [ ]:

```
xg_model = XGBClassifier()
xg_model.fit(trainX, trainY)
```

Out[ ]:

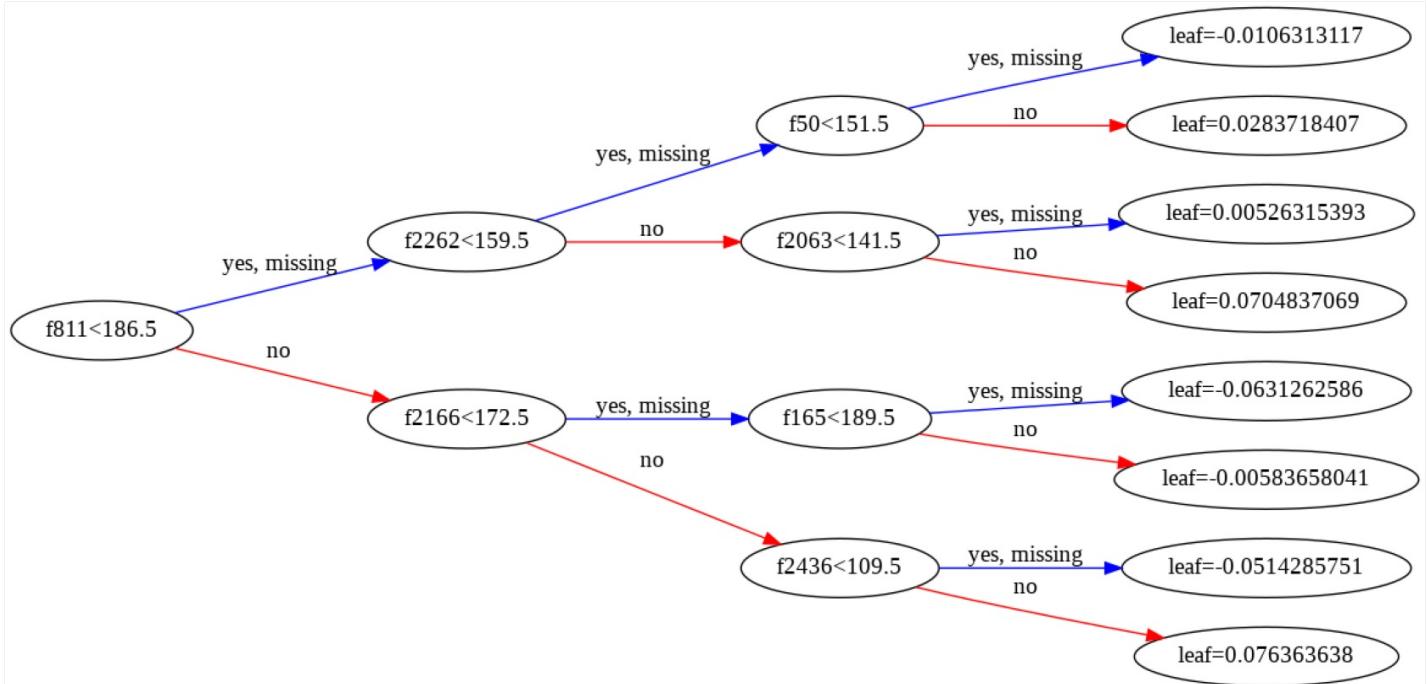
```
XGBClassifier(objective='multi:softprob')
```

### Visualizing Individual Trees

In [ ]:



```
plot_tree(xg_model, num_trees=0, rankdir='LR')
fig = plt.gcf()
fig.set_size_inches(30, 30)
```



## Accuracy Score

In [ ]:

```
test_predictions = xg_model.predict(testX)
test_accuracy = accuracy_score(testY, test_predictions)
print("Test Accuracy: %.2f%%" % (test_accuracy * 100.0))
```

Test Accuracy: 67.17%

In [ ]:

```

#@title Function to plot confusion matrix
def plot_confusion_matrix(cm, classes,
                           normalize=False,
                           title='Confusion matrix',
                           cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

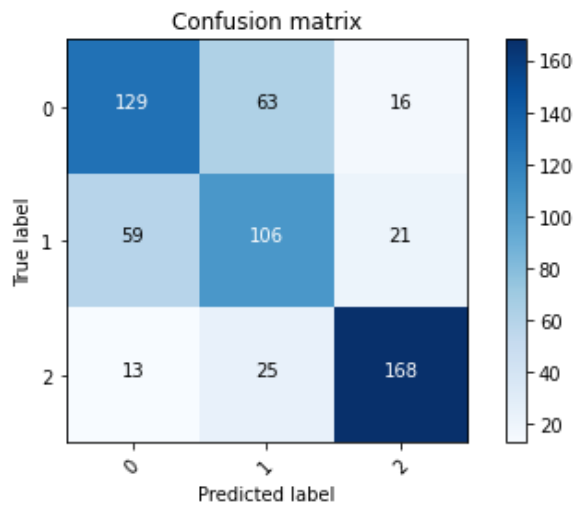
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

In [ ]:

```

confusion_matrix(testY, test_predictions)
target_names = ['0', '1', '2']
#Pass Actual & Predicted values to confusion_matrix()
cm = confusion_matrix(testY, test_predictions)
plt.figure()
plot_confusion_matrix(cm, classes=target_names)
```

```
plt.show()
```



## CO5:COMPARSION OF PERFORMANCE

### Results and Discussion

***KNN - Accuracy: 47.33 %***

***Decision Tree - Accuracy: 50.5%***

***SVC - Accuracy: 62.67 %***

***Voting Classifier with HardVoting-60.33%***

***Voting Classifier with SoftVoting-57.50%***

***Bagging Classier- 63.00%***

***Kfold-62.17%***

***Gradient Boosting-58.17%***

***AdaBoost-57.05%***

***RandomClassifier-61.50%***

***ANN-33.8%***

***Perceptron Model-52.71%***

***Homogenous Ensembling technique-49.50%***

***XGBOOST- 67.17%***

## CONCLUSION OF PROJECT OUTCOME

Ensemble learning is a general approach to machine learning that seeks better predictive performance by combining the predictions from multiple models. There are two main reasons to use an ensemble over a single model, and they are:

**Performance:** An ensemble can make better predictions and achieve better performance than any single contributing model.

***Robustness: An ensemble reduces the spread or dispersion of the predictions and model performance.***

Ensembles are used to achieve better predictive performance on a predictive modeling problem than a single predictive model. The way this is achieved can be understood as the model reducing the variance component of the prediction error by adding bias. We have implemented all the techniques and understood the use case of each to classify and the three classes (dogs, cats, pandas) are classified with least error in the XGBoost Model which has the highest accuracy of 67.17%.