

Machine Learning Assignment

Ujjwal Sharma - BT18CSE021
Chinmay Hattewar - BT18CSE055

CLASSIFICATION PROBLEM

(Predict whether a mammogram mass is benign or malignant)

MODEL USED :

1. DECISION TREES
2. RANDOM FORESTS
3. K-MEANS AND KNN
4. SUPPORT VECTOR MACHINE
5. NAIVE BAYES
6. ARTIFICIAL NEURAL NETWORK

DATASET

—

Mammographic Mass Data

A. Original owners of database:

Prof. Dr. Rüdiger Schulz-Wendtland
Institute of Radiology, Gynaecological Radiology,
University Erlangen-Nuremberg
91054 Erlangen, Germany

B. Donor of database:

Matthias Elter
Fraunhofer Institute for Integrated Circuits (IIS)
Image Processing and Medical Engineering Department (BMT)
Am Wolfsmantel 33
91058 Erlangen, Germany
matthias.elter@iis.fraunhofer.de
(49) 9131-7767327

Relevant Information

Mammography is the most effective method for breast cancer screening available today. However, the low positive predictive value of breast biopsy resulting from mammogram interpretation leads to approximately 70% unnecessary biopsies with benign outcomes. To reduce the high number of unnecessary breast biopsies, several computer-aided diagnosis (CAD) systems have been proposed in the last years. These systems help physicians in their decision to perform a breast biopsy on a suspicious lesion seen in a mammogram or to perform a short term follow-up examination instead.

This data set can be used to predict the severity (benign or malignant) of a mammographic mass lesion from BI-RADS attributes and the patient's age. It contains a BI-RADS assessment, the patient's age and three BI-RADS attributes together with the ground truth (the severity field) for 516 benign and 445 malignant masses that have been identified on full field digital mammograms collected at the Institute of Radiology of the University Erlangen-Nuremberg between 2003 and 2006. Each instance has an associated BI-RADS assessment ranging from 1 (definitely benign) to 5 (highly suggestive of malignancy) assigned in a double-review process by physicians. Assuming that all cases with BI-RADS assessments greater or equal a given value (varying from 1 to 5), are malignant and the other cases benign, sensitivities and associated specificities can be calculated. These can be an indication of how well a CAD system performs compared to the radiologists.

Information Regarding Dataset

1. Number of Instances: 961

2. Number of Attributes: 6 (1 goal field, 1 non-predictive, 4 predictive attributes)

3. Attribute Information:

1. BI-RADS assessment: 1 to 5 (ordinal)

2. Age: patient's age in years (integer)

3. Shape: mass shape: round=1 oval=2 lobular=3 irregular=4 (nominal)

4. Margin: mass margin: circumscribed=1 microlobulated=2 obscured=3 ill-defined=4 spiculated=5 (nominal)

5. Density: mass density high=1 iso=2 low=3 fat-containing=4 (ordinal)

6. Severity: benign=0 or malignant=1 (binominal)

Information Regarding Dataset

4. Missing Attribute Values: Yes

- BI-RADS assessment: 2

- Age: 5

- Shape: 31

- Margin: 48

- Density: 76

- Severity: 0

5. Class Distribution: benign: 516; malignant: 445

CLASSIFICATION MODELS

—

DECISION TREE

API USED FROM SKLEARN

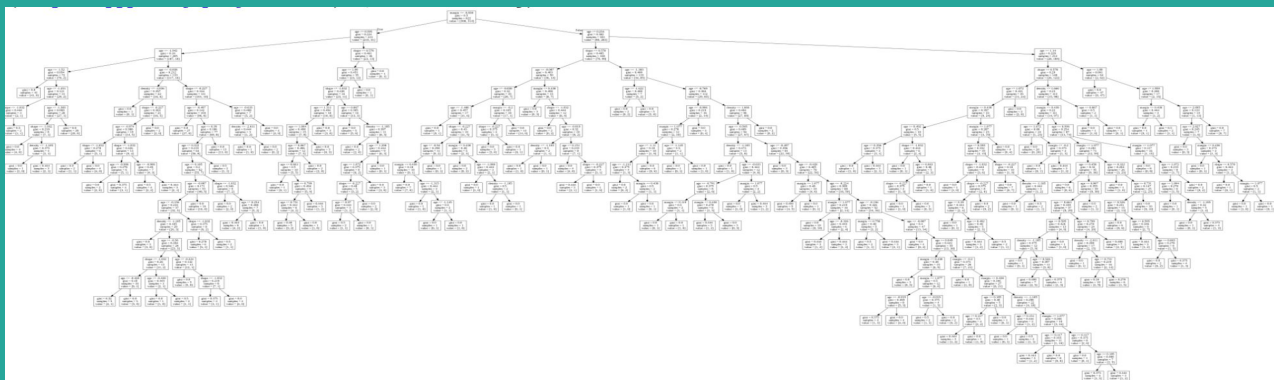
```
▶ from sklearn.tree import DecisionTreeClassifier

decisionTreeClassifier = DecisionTreeClassifier(random_state=1)

# Trainnng the classifier on the training set
decisionTreeClassifier.fit(training_inputs, training_classes)

↳ DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                        max_depth=None, max_features=None, max_leaf_nodes=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=1, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, presort='deprecated',
                        random_state=1, splitter='best')
```

Final Decision Tree



Accuracy Of Decision Tree

Train-Test with 75% split :

```
▶ decisionTreeClassifier.score(testing_inputs, testing_classes)
```

```
📄 0.7355769230769231
```

K-Cross Validation with $k = 10$

```
crossValidationScores = cross_val_score(decisionTreeClassifierKCross, all_features_scaled, all_classes, cv=10)
```

```
crossValidationScores.mean()
```

```
📄 0.7373493975903613
```

K-Cross Validation with $k = 5$

```
crossValidationScores = cross_val_score(decisionTreeClassifierKCross, all_features_scaled, all_classes, cv=5)
```

```
crossValidationScores.mean()
```

```
0.7253012048192771
```

DECISION TREE

Self Implemented

Dataset Used - Play Tennis

<https://www.kaggle.com/fredericobreno/play-tennis>

Test-Train split 70%

▶ data.head()



| | day | outlook | temp | humidity | wind | play |
|---|-----|----------|------|----------|--------|------|
| 0 | D1 | Sunny | Hot | High | Weak | No |
| 1 | D2 | Sunny | Hot | High | Strong | No |
| 2 | D3 | Overcast | Hot | High | Weak | Yes |

```
[ ] tree = ID3(trainData, 'play')
    tree
```

```
{'humidity': {'High': {'outlook': {'Overcast': 'Yes',
    'Rain': {'wind': {'Strong': 'No', 'Weak': 'Yes'}}},
    'Sunny': 'No'}}},
    'Normal': 'Yes'}}
```



```
accuracy = evaluate(tree , testData , 'play')
print("Accuracy of Decision Tree : " , accuracy)
```



```
Accuracy of Decision Tree : 0.8
```

Advantages Of Decision Tree

- 1. Clear Visualization:** The algorithm is simple to understand, interpret and visualize as the idea is mostly used in our daily lives. Output of a Decision Tree can be easily interpreted by humans.
- 2. Simple and easy to understand:** Decision Tree looks like simple **if-else statements** which are very easy to understand.
- 3.** Decision Tree can be used for both **classification and regression problems**.
- 4.** Decision Tree can handle both **continuous and categorical variables**.
- 5. No feature scaling required:** No feature scaling (standardization and normalization) required in case of Decision Tree as it uses rule based approach instead of distance calculation.
- 6. Handles non-linear parameters efficiently:** Non linear parameters don't affect the performance of a Decision Tree unlike curve based algorithms. So, if there is high nonlinearity between the independent variables, Decision Trees may outperform as compared to other curve based algorithms.
- 7.** Decision Tree can automatically **handle missing values**.
- 8.** Decision Tree is usually **robust to outliers** and can handle them automatically.
- 9. Less Training Period:** Training period is less as compared to Random Forest because it generates only one tree unlike forest of trees in the Random Forest.

Disadvantages Of Decision Tree

1. Overfitting: This is the main problem of the Decision Tree. It generally leads to overfitting of the data which ultimately leads to wrong predictions. In order to fit the data (even noisy data), it keeps generating new nodes and ultimately the tree becomes too complex to interpret. In this way, it loses its generalization capabilities. It performs very well on the trained data but starts making a lot of mistakes on the unseen data.

2. High variance: As mentioned in point 1, Decision Tree generally leads to the overfitting of data. Due to the overfitting, there are very high chances of high variance in the output which leads to many errors in the final estimation and shows high inaccuracy in the results. In order to achieve zero bias (overfitting), it leads to high variance.

3. Unstable: Adding a new data point can lead to re-generation of the overall tree and all nodes need to be recalculated and recreated.

4. Affected by noise: Little bit of noise can make it unstable which leads to wrong predictions.

5. Not suitable for large datasets: If data size is large, then one single tree may grow complex and lead to overfitting. So in this case, we should use Random Forest instead of a single Decision Tree.

Random Forest

Ensemble Approach

API USED FROM SKLEARN

TOTAL ESTIMATORS = 100

```
▶ from sklearn.ensemble import RandomForestClassifier

randomForestClassifier100 = RandomForestClassifier(n_estimators=100)

randomForestClassifier100.fit(training_inputs, training_classes)
randomForestClassifier100.score(testing_inputs , testing_classes)
```

☞ 0.7692307692307693

K-Cross Validation

```
▶ from sklearn.ensemble import RandomForestClassifier

randomForestClassifier100 = RandomForestClassifier(n_estimators=100)
randomForestClassifier100_10Cross = cross_val_score(randomForestClassifier100, all_features_scaled, all_classes, cv=10)

randomForestClassifier100_10Cross.mean()
```

☞ 0.755421686746988

API USED FROM SKLEARN

TOTAL ESTIMATORS = 200

```
randomForestClassifier200 = RandomForestClassifier(n_estimators=200)

randomForestClassifier200.fit(training_inputs, training_classes)
randomForestClassifier200.score(testing_inputs , testing_classes)

0.7596153846153846
```

K-Cross Validation

```
from sklearn.ensemble import RandomForestClassifier

randomForestClassifier200 = RandomForestClassifier(n_estimators=200)
randomForestClassifier200_10Cross = cross_val_score(randomForestClassifier100, all_features_scaled, all_classes, cv=10)

randomForestClassifier200_10Cross.mean()

0.7566265060240964
```

Advantages Of Random Forest

1. Random Forest is based on the **bagging** algorithm and uses **Ensemble Learning** technique. It creates as many trees on the subset of the data and combines the output of all the trees. In this way it **reduces overfitting** problem in decision trees and also **reduces the variance** and therefore **improves the accuracy**.
2. Random Forest can be used to **solve both classification as well as regression problems**.
3. Random Forest works well with both **categorical and continuous variables**.
4. Random Forest can automatically **handle missing values**.
5. **No feature scaling required:** No feature scaling (standardization and normalization) required in case of Random Forest as it uses rule based approach instead of distance calculation.
6. **Handles non-linear parameters efficiently:** Non linear parameters don't affect the performance of a Random Forest unlike curve based algorithms. So, if there is high non-linearity between the independent variables, Random Forest may outperform as compared to other curve based algorithms.
7. Random Forest can automatically **handle missing values**.
8. Random Forest is usually **robust to outliers** and can handle them automatically.
9. Random Forest algorithm is very **stable**. Even if a new data point is introduced in the dataset, the overall algorithm is not affected much since the new data may impact one tree, but it is very hard for it to impact all the trees.
10. Random Forest is comparatively **less impacted by noise**.

Disadvantages Of Random Forest

1. Complexity: Random Forest creates a lot of trees (unlike only one tree in case of decision tree) and combines their outputs. By default, it creates 100 trees in Python sklearn library. To do so, this algorithm requires much more computational power and resources. On the other hand decision tree is simple and does not require so much computational resources.

2. Longer Training Period: Random Forest require much more time to train as compared to decision trees as it generates a lot of trees (instead of one tree in case of decision tree) and makes decision on the majority of votes.

K-Means And KNN

API USED FROM SKLEARN for KMEANS

```
▶ from sklearn.cluster import KMeans
  from sklearn.preprocessing import scale
  from numpy import random
  kmeans = KMeans(n_clusters=2)
  scaledFeatures = scale(all_features)
  kmeans.fit(scaledFeatures)

[→ KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
          n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
          random_state=None, tol=0.0001, verbose=0)
```

Accuracy of K-Means

```
▶ from sklearn.metrics import accuracy_score  
print("Accuracy of K-means is : ",accuracy_score(kmeans.labels_ , all_classes))
```

```
☐→ Accuracy of K-means is : 0.7975903614457831
```

Here labels actually don't have any meaning unless i specify
So in this case i have specified 0 as No and 1 as Yes

KNN

Self Implemented

```
loadDataset('Imputed_for_knn.csv', split, trainingSet, testSet)
print('Train set: ' + repr(len(trainingSet)))
print('Test Set: ' + repr(len(testSet)))

# Generate Predictions
predictions = []
k = 3
for x in range(len(testSet)):
    neighbors = getNeighbors(trainingSet, testSet[x], k)
    res = getResponse(neighbors)
    predictions.append(res)

print("predicted = " + repr(res) + ", Actual = " + repr(testSet[x][-1]))
```

Accuracy of KNN

For $k = 3$

```
accuracy = getAcuuracy(testSet, predictions)
print("Accuracy: " + repr(accuracy) + " %")

Accuracy: 58.74125874125874 %
```

For $k = 5$

```
accuracy = getAcuuracy(testSet, predictions)
print("Accuracy: " + repr(accuracy) + " %")

Accuracy: 61.111111111111114 %
```

For $k = 7$

```
accuracy = getAcuuracy(testSet, predictions)
print("Accuracy: " + repr(accuracy) + " %")

Accuracy: 60.71428571428571 %
```


Advantages Of K-Means

1. Relatively simple to implement.
2. Scales to large data sets.
3. Guarantees convergence.
4. Can warm-start the positions of centroids.
5. Easily adapts to new examples.
6. Generalizes to clusters of different shapes and sizes, such as elliptical clusters.

Disadvantages Of K-Means

1. **Choosing K manually.**
2. **Being dependent on initial values.**
 - a. For a low k, you can mitigate this dependence by running k-means several times with different initial values and picking the best result. As k increases, you need advanced versions of k-means to pick better values of the initial centroids (called **k-means seeding**).
3. **Clustering data of varying sizes and density.**
 - a. k-means has trouble clustering data where clusters are of varying sizes and density. To cluster such data, you need to generalize k-means .
4. **Clustering outliers.**
 - a. Centroids can be dragged by outliers, or outliers might get their own cluster instead of being ignored. Consider removing or clipping outliers before clustering.
5. **Scaling with number of dimensions.**
 - a. As the number of dimensions increases, a distance-based similarity measure converges to a constant value between any given examples. Reduce dimensionality either by using **PCA** on the feature data, or by using “spectral clustering” to modify the clustering algorithm as explained below.

SUPPORT VECTOR MACHINE

API USED FROM SKLEARN

```
[ ] from sklearn import svm

C = 1.0
kernel = 'linear'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputs, training_classes)
```

Accuracy of SVM Linear Kernel

```
from sklearn import svm

C = 1.0
kernel = 'linear'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputs, training_classes)

print("Test-Train Split Score : " , svc.score(testing_inputs , testing_classes))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
Test-Train Split Score : 0.7692307692307693
K-Cross Score (k = 10) : 0.7975903614457832
K-Cross Score (k = 20) : 0.8036004645760745
```

Accuracy of SVM RBF Kernel

```
▶ from sklearn import svm

C = 1.0
kernel = 'rbf'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputs, training_classes)

print("Test-Train Split Score : " , svc.score(testing_inputs , testing_classes))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
☞ Test-Train Split Score : 0.7788461538461539
   K-Cross Score (k = 10) : 0.8012048192771084
   K-Cross Score (k = 20) : 0.8046457607433218
```

Accuracy of SVM Sigmoid Kernel

```
▶ from sklearn import svm

C = 1.0
kernel = 'sigmoid'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputs, training_classes)

print("Test-Train Split Score : " , svc.score(testing_inputs , testing_classes))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
↳ Test-Train Split Score : 0.7067307692307693
   K-Cross Score (k = 10) : 0.7457831325301204
   K-Cross Score (k = 20) : 0.7364401858304298
```

Accuracy of SVM Polynomial Kernel

```
▶ from sklearn import svm

C = 1.0
kernel = 'poly'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputs, training_classes)

print("Test-Train Split Score : " , svc.score(testing_inputs , testing_classes))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, all_features_scaled, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
☞ Test-Train Split Score : 0.75
   K-Cross Score (k = 10) : 0.7903614457831326
   K-Cross Score (k = 20) : 0.7853077816492452
```


Advantages Of Support Vector Machine

- 1. Regularization capabilities:** SVM has L2 Regularization feature. So, it has good generalization capabilities which prevent it from over-fitting.
- 2. Handles non-linear data efficiently:** SVM can efficiently handle non-linear data using Kernel trick.
- 3. Solves both Classification and Regression problems:** SVM can be used to solve both classification and regression problems. SVM is used for classification problems while **SVR (Support Vector Regression)** is used for regression problems.
- 4. Stability:** A small change to the data does not greatly affect the hyperplane and hence the SVM. So the SVM model is stable.

Disadvantages Of Support Vector Machine

- 1. Choosing an appropriate Kernel function is difficult:** Choosing an appropriate Kernel function (to handle the non-linear data) is not an easy task. It could be tricky and complex. In case of using a high dimension Kernel, you might generate too many support vectors which reduce the training speed drastically.
- 2. Extensive memory requirement:** Algorithmic complexity and memory requirements of SVM are very high. You need a lot of memory since you have to store all the support vectors in the memory and this number grows abruptly with the training dataset size.
- 3. Requires Feature Scaling:** One must do feature scaling of variables before applying SVM.
- 4. Long training time:** SVM takes a long training time on large datasets.
- 5. Difficult to interpret:** SVM model is difficult to understand and interpret by human beings unlike Decision Trees.

NAIVE BAYES

API USED FROM SKLEARN

```
scaler = preprocessing.MinMaxScaler()  
#scaling the data b/w minimum and maximum  
  
all_featuresMinMax = scaler.fit_transform(all_features)
```



```
from sklearn.naive_bayes import MultinomialNB  
  
naiveBayesClassifier = MultinomialNB()  
  
naiveBayesClassifier.fit(training_inputsMinMax , training_classesMinMax)
```

Accuracy of Naive Bayes

```
print("Test-Train Split Score : " , naiveBayesClassifier.score( testing_inputsMinMax, testing_classesMinMax))

naiveBayesClassifierKCross = MultinomialNB()

naiveBayesClassifierKCrossScore10 = cross_val_score(naiveBayesClassifierKCross, all_featuresMinMax, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , naiveBayesClassifierKCrossScore10.mean())

naiveBayesClassifierKCrossScore20 = cross_val_score(naiveBayesClassifierKCross, all_featuresMinMax, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , naiveBayesClassifierKCrossScore20.mean())
```

```
Test-Train Split Score : 0.7548076923076923
K-Cross Score (k = 10) : 0.7855421686746988
K-Cross Score (k = 20) : 0.7829558652729385
```

Advantages Of Naive Bayes

1. When assumption of independent predictors holds true, a Naive Bayes classifier performs better as compared to other models.
2. Naive Bayes requires a small amount of training data to estimate the test data. So, the training period is less.
3. Naive Bayes is also easy to implement.

Disadvantages Of Naive Bayes

1. Main imitation of Naive Bayes is the **assumption of independent predictors**. Naive Bayes implicitly assumes that all the attributes are mutually independent. In real life, it is almost impossible that we get a set of predictors which are completely independent.
2. If categorical variable has a category in test data set, which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as **Zero Frequency**. To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called **Laplace estimation**.

ARTIFICIAL NEURAL NETWORKS

API USED FROM KERAS

(TENSORFLOW)



```
import keras
from keras.models import Sequential
from keras.layers import Dense , Activation , Dropout
import tensorflow
```

API USED FROM KERAS

(TENSORFLOW)

ADDING OPTIMIZER , DROPOUT AND REGULARISATION (L2) TO AVOID OVERFITTING

```
model = Sequential()

model.add(Dense(10 , input_shape=inputSize , kernel_regularizer='l2'))
model.add(Activation('relu'))
model.add(Dropout(0.3))

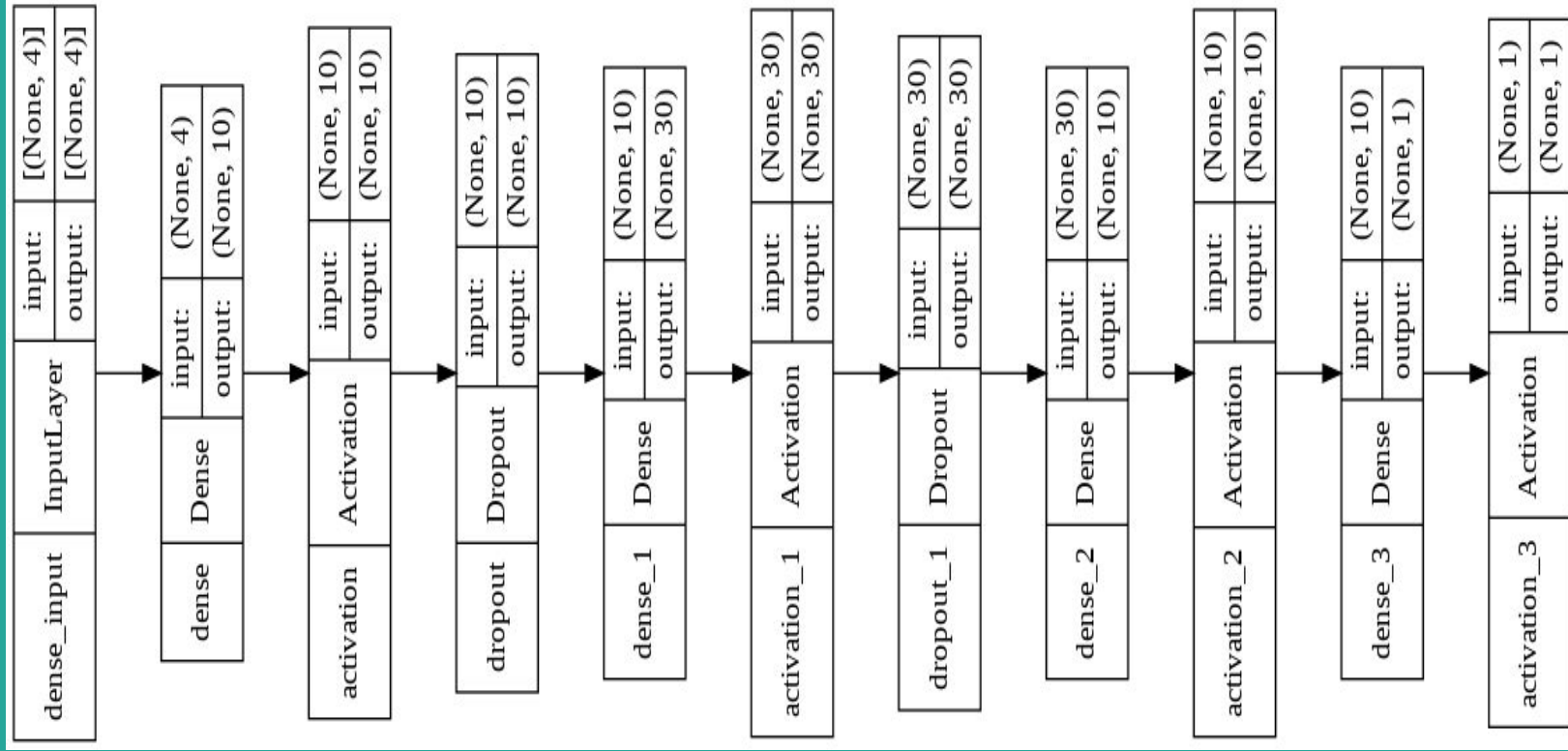
model.add(Dense(30 , kernel_regularizer='l2'))
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(10 , kernel_regularizer='l2'))
model.add(Activation('relu'))

model.add(Dense(1 , kernel_regularizer='l2'))
model.add(Activation('sigmoid'))

optimizer = tensorflow.keras.optimizers.RMSprop()
model.compile(loss='binary_crossentropy' , optimizer=optimizer , metrics = ['accuracy'])
```

Visual Representation Of Model



Accuracy after 20 epochs

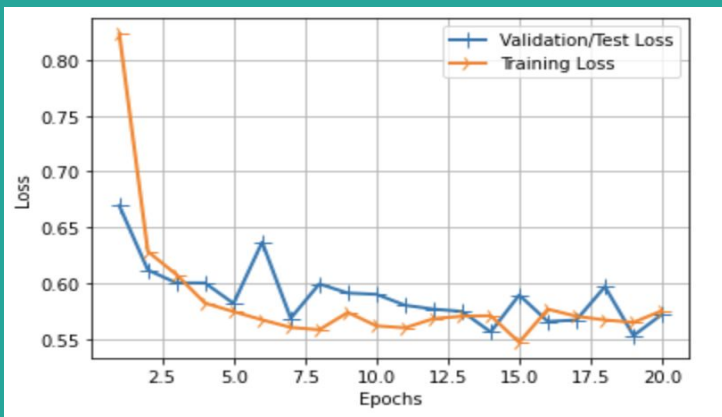
Test-Train Split 66% , batchSize = 1

```
▶ epochs = 20  
batch = 1  
history = model.fit(training_inputs , training_classes , epochs = epochs  
                    ,batch_size=batch , verbose=1 ,  
                    validation_data = (testing_inputs , testing_classes))
```

Epoch 20/20

547/547 [=====] - 1s 2ms/step - loss: 0.5755 - accuracy: 0.8263 - val_loss: 0.5723 - val_accuracy: 0.7915

Graph of Loss VS Epochs Of the Model



Graph of Accuracy VS Epochs Of the Model



Accuracy after 50 epochs

Test-Train Split 66% , batchSize = 4

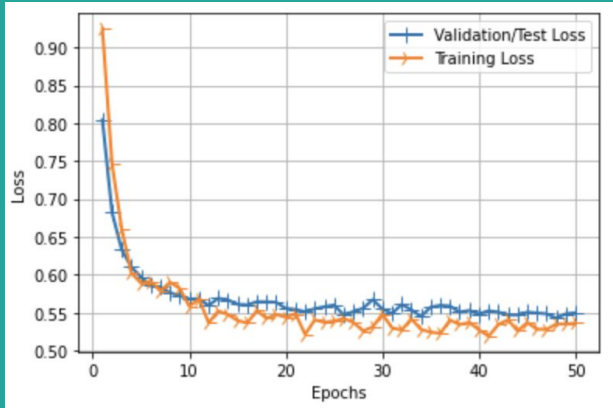


```
epochs = 50
batch = 4
history = model.fit(training_inputs , training_classes , epochs = epochs
                    ,batch_size=batch , verbose=1 ,
                    validation_data = (testing_inputs , testing_classes))
```

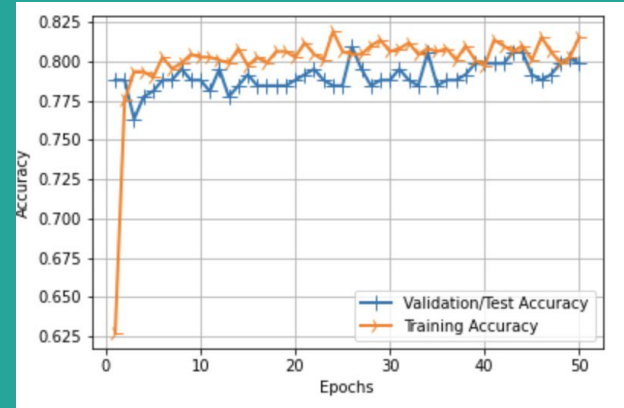
Epoch 50/50

137/137 [=====] - 0s 2ms/step - loss: 0.5362 - accuracy: 0.8154 - val_loss: 0.5498 - val_accuracy: 0.7986

Graph of Loss VS Epochs Of the Model



Graph of Accuracy VS Epochs Of the Model



K-Cross Validation



```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

# Wrap our Keras model
estimator = KerasClassifier(build_fn=createModel, epochs=50, verbose=0)
# Now we can use scikit_learn's cross_val_score to evaluate this model
CrossValidationScoresNeuralNet = cross_val_score(estimator, all_features_scaled, all_classes, cv=10)
print('\n\n\n')

print("Average Accuracy with k = %d is : "%10,CrossValidationScoresNeuralNet.mean())
```

↳ /usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: DeprecationWarning: KerasClassifier is deprecated after removing the cwd from sys.path.

Average Accuracy with k = 10 is : 0.8036144614219666

Artificial Neural Network

(Self Implemented with backpropagation upto 3 layers)

- Class representing a linear layer having n_inp neurons as input and n_out neurons as output

```
class Linearlayer():  
    def __init__(self,n_inp,n_out):  
        self.weights = np.random.randn(n_inp,n_out)  
        self.bias = np.zeros((1,n_out))  
    def forward(self,inputs):  
        self.output = np.dot(inputs,self.weights) + self.bias
```

- Neural Network

```
class NeuralNet():  
    def __init__(self,n_inp,n_out,alpha):  
        self.inp = n_inp  
        self.out = n_out  
        self.hidd_no1 = 60  
        self.hidd_no2 = 80  
        self.alpha = alpha  
        self.error = 1  
        self.layer1 = Linearlayer(n_inp,self.hidd_no1)  
        self.layer2 = Linearlayer(self.hidd_no1,self.hidd_no2)  
        self.layer3 = Linearlayer(self.hidd_no2,n_out)  
  
    # sigmoid activation function  
    def act_fun(self,x):  
        return 1/(1+exp(-x))  
    # derivation of sigmoid function  
    def der_act_fun(self,x):  
        x = self.act_fun(x)  
        return x*(1-x)  
    # forward pass through the network  
    def forward(self,input_set):  
        self.input_set = input_set  
        self.layer1.forward(input_set)  
        # applying activation function after layer1 pass  
        self.inp_hidden1 = self.act_fun(self.layer1.output)  
        # applying activation function after layer2 pass  
        self.layer2.forward(self.inp_hidden1)  
        self.inp_hidden2 = self.act_fun(self.layer2.output)  
        self.layer3.forward(self.inp_hidden2)  
        # applying activation function after layer3 pass  
        self.fout = self.act_fun(self.layer3.output)
```

```
def learn(self,input_set,output_set):  
    nnout = self.forward(input_set)  
    print("output is - ",nnout)  
    self.error = 0  
    for i in range(len(output_set)):  
        self.error+=(output_set[i]-nnout[0][i])**2  
    self.error/=2  
    # print("error - ",self.error)  
  
    # Backpropogating the error  
    self.backpropgatel1(output_set)  
    self.backpropgatel2()  
    self.backpropgatel3()  
  
def predict(self , features):  
    lst = list(self.forward(features)[0])  
    index = lst.index(max(lst))  
    return index
```

Artificial Neural Network

(Self Implemented with backpropagation upto 3 layers)

Training the neural network

```
▶ size = training_inputs.shape[0]
error = 0
count = 0
lmt = size//4
for row,label in zip(training_inputs , training_classes):
    net.learn(row , label)
    error += net.error
    count+=1
    if count%lmt==0:
        print(count/size*100 , '%')
        print("error = ",error/count)
```


Accuracy of self implemented Neural Network

Learning Rate = 0.1 , 10 Epochs

▼ Checking the accuracy of the model

learning rate = 0.1

✓
0s

```
▶ count = 0
  correct = 0
  for row,label in zip(testing_inputs , testing_classes):
      lst = list(label)
      index = lst.index(max(lst))
      value = net.predict(row)
      count+=1
      if index == value:
          correct+=1
  print("Accuracy of the model : " , correct/count*100)
```

☞ Accuracy of the model : 77.88461538461539

Accuracy of self implemented Neural Network

Learning Rate = 0.01 , 10 Epochs

▼ Checking the accuracy of model

learning rate = 0.01

✓
0s

```
▶ count = 0
  correct = 0
  for row,label in zip(testing_inputs , testing_classes):
      lst = list(label)
      index = lst.index(max(lst))
      value = net2.predict(row)
      count+=1
      if index == value:
          correct+=1
  print("Accuracy of the model : " , correct/count*100)
```

☞ Accuracy of the model : 57.21153846153846

Accuracy of self implemented Neural Network

Learning Rate = 0.3 , 10 Epochs

▼ Checking the accuracy of the model

learning rate = 0.3

✓
0s



```
count = 0
correct = 0
for row,label in zip(testing_inputs , testing_classes):
    lst = list(label)
    index = lst.index(max(lst))
    value = net3.predict(row)
    count+=1
    if index == value:
        correct+=1
print("Accuracy of the model : " , correct/count*100)
```

☞ Accuracy of the model : 77.40384615384616

Advantages Of Artificial Neural Networks

1. Problems in ANN are represented by attribute-value pairs.
2. ANNs are used for problems having the target function, the output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes.
3. ANN learning methods are quite robust to noise in the training data. The training examples may contain errors, which do not affect the final output.
4. It is used where the fast evaluation of the learned target function required.
5. ANNs can bear long training times depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.

Disadvantages Of Artificial Neural Networks

1. Hardware Dependence:
 - a. Artificial Neural Networks require processors with parallel processing power, by their structure.
 - b. For this reason, the realization of the equipment is dependent.
2. Unexplained functioning of the network:
 - a. This the most important problem of ANN.
 - b. When ANN gives a probing solution, it does not give a clue as to why and how.
 - c. This reduces trust in the network.
3. Assurance of proper network structure:
 - a. There is no specific rule for determining the structure of artificial neural networks.
 - b. The appropriate network structure is achieved through experience and trial and error.

Disadvantages Of Artificial Neural Networks

1. The difficulty of showing the problem to the network:
 - a. ANNs can work with numerical information.
 - b. Problems have to be translated into numerical values before being introduced to ANN.
 - c. The display mechanism to be determined will directly influence the performance of the network.
 - d. This is dependent on the user's ability.
2. The duration of the network is unknown:
 - a. The network is reduced to a certain value of the error on the sample means that the training has been completed.
 - b. The value does not give us optimum results.

Some Other Techniques

—

Principal Component Analysis (PCA)

The **principal components** of a collection of points in a **real coordinate space** are a sequence of **unit vectors**, where the i -th vector is the direction of a line that best fits the data while being **orthogonal** to the first $i-1$ vectors. Here, a best-fitting line is defined as one that minimizes the average squared **distance from the points to the line**. These directions constitute an **orthonormal basis** in which different individual dimensions of the data are **linearly uncorrelated**. **Principal component analysis (PCA)** is the process of computing the principal components and using them to perform a **change of basis** on the data, sometimes using only the first few principal components and ignoring the rest.

WIKIPEDIA

Simple Definition

PCA is a dimensionality reduction technique; it lets you distill multi-dimensional data down to fewer dimensions, selecting new dimensions that preserve variance in the data as best it can.

API USED FROM SKLEARN

```
✓ [31] from sklearn.decomposition import PCA
```

0s

▼ With 2 components

```
✓ [32] ▶ pca2 = PCA(n_components=2, whiten=True).fit(all_features)  
allFeaturesPCA2 = pca2.transform(all_features)
```

0s

▼ lets see how much variance we preserved

```
✓ [33] print(pca2.explained_variance_ratio_)  
print(sum(pca2.explained_variance_ratio_))
```

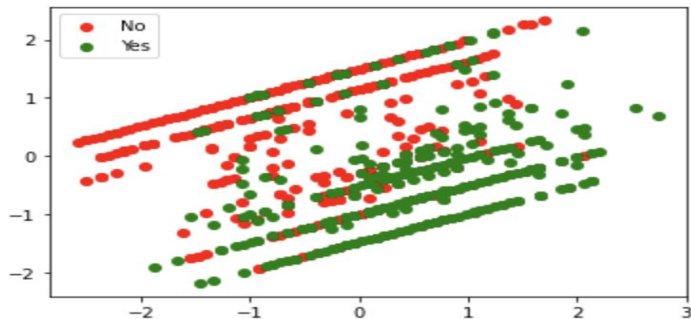
0s

```
[0.98424958 0.01295367]  
0.997203256835024
```

Plotting the features

```
%matplotlib inline
import pylab as pl
from pylab import *
from itertools import cycle

colors = cycle('rgb')
target_ids = range(2)
pl.figure()
for i, c, label in zip(target_ids, colors, ["No" , "Yes"]):
    pl.scatter(allFeaturesPCA2[all_classes == i, 0], allFeaturesPCA2[all_classes == i, 1],
               c=c, label=label)
pl.legend()
pl.show()
```



K-Means with 2 Components

▼ Kmeans on PCA with n = 2

✓
0s



```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from numpy import random
```

```
kmeansPCA2 = KMeans(n_clusters=2)
```

```
scaledFeaturesPCA2 = scale(allFeaturesPCA2)
```

```
kmeansPCA2.fit(scaledFeaturesPCA2)
```

```
↳ KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
          n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
          random_state=None, tol=0.0001, verbose=0)
```

✓
0s

```
[36] from sklearn.metrics import accuracy_score
      print("Accuracy of K-means is : ",accuracy_score(kmeansPCA2.labels_ , all_classes))
```

```
Accuracy of K-means is : 0.7939759036144578
```

SVM with 2 Components

▼ SVM with 2 components

✓
0s



```
from sklearn import svm

C = 1.0
kernel = 'linear'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputsPCA2, training_classesPCA2)

print("Test-Train Split Score : " , svc.score(testing_inputsPCA2 , testing_classesPCA2))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, scaledFeaturesPCA2, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, scaledFeaturesPCA2, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
☞ Test-Train Split Score : 0.7548076923076923
   K-Cross Score (k = 10) : 0.7915662650602411
   K-Cross Score (k = 20) : 0.7901277584204414
```

API USED FROM SKLEARN

▼ With 3 components

```
✓ [37]  
0s      pca3 = PCA(n_components=3, whiten=True).fit(all_features)  
      allFeaturesPCA3 = pca3.transform(all_features)
```

▼ Lets see how much variance we captured

```
✓ [38] print(pca3.explained_variance_ratio_)  
0s      print(sum(pca3.explained_variance_ratio_))  
  
[0.98424958 0.01295367 0.00224542]  
0.9994486808624999
```

K-Means with 3 Components

▼ Kmeans with PCA n = 3

```
✓ [68] from sklearn.cluster import KMeans  
0s      import matplotlib.pyplot as plt  
      from sklearn.preprocessing import scale  
      from numpy import random  
  
      kmeansPCA3 = KMeans(n_clusters=2)  
  
      scaledFeaturesPCA3 = scale(allFeaturesPCA3)  
  
      kmeansPCA3.fit(scaledFeaturesPCA3)  
  
      KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,  
              n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',  
              random_state=None, tol=0.0001, verbose=0)  
  
✓ [69] from sklearn.metrics import accuracy_score  
0s      print("Accuracy of K-means is : ", accuracy_score(1-kmeansPCA3.labels_ , all_classes))  
  
      Accuracy of K-means is : 0.7710843373493976
```

SVM with 3 Components

➤ SVM with 3 components

✓
0s

```
from sklearn import svm

C = 1.0
kernel = 'linear'
svc = svm.SVC(kernel=kernel, C=C)

svc.fit(training_inputsPCA3, training_classesPCA3)

print("Test-Train Split Score : " , svc.score(testing_inputsPCA3 , testing_classesPCA3))

svcKCross = svm.SVC(kernel=kernel, C=C)
svcKCrossScores = cross_val_score(svc, scaledFeaturesPCA3, all_classes, cv=10)
print("K-Cross Score (k = %d) : "%10 , svcKCrossScores.mean())
svcKCrossScores = cross_val_score(svc, scaledFeaturesPCA3, all_classes, cv=20)
print("K-Cross Score (k = %d) : "%20 , svcKCrossScores.mean())
```

```
Test-Train Split Score : 0.7692307692307693
K-Cross Score (k = 10) : 0.7975903614457832
K-Cross Score (k = 20) : 0.8036004645760745
```


Ensemble learning

Ensemble learning is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular **computational intelligence** problem. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model, or reduce the likelihood of an unfortunate selection of a poor one. Other applications of ensemble learning include assigning a confidence to the decision made by the model, selecting optimal (or near optimal) features, data fusion, incremental learning, nonstationary learning and error-correcting. This article focuses on classification related applications of ensemble learning, however, all principle ideas described below can be easily generalized to function approximation or prediction type problems as well.

TYPES OF ENSEMBLE LEARNING

Bagging

Bagging, which stands for *bootstrap aggregating*, is one of the earliest, most intuitive and perhaps the simplest ensemble based algorithms, with a surprisingly good performance (Breiman 1996). Diversity of classifiers in bagging is obtained by using bootstrapped replicas of the training data.

Boosting

Similar to bagging, **boosting** also creates an ensemble of classifiers by resampling the data, which are then combined by majority voting. However, in boosting, resampling is strategically geared to provide the most informative training data for each consecutive classifier


AdaBoost

Arguably the best known of all ensemble-based algorithms, **AdaBoost** (Adaptive Boosting) extends boosting to multi-class and regression problems

Stacked Generalization

In Wolpert's **stacked generalization** (or **stacking**), an ensemble of classifiers is first trained using bootstrapped samples of the training data, creating *Tier 1 classifiers*, whose outputs are then used to train a *Tier 2 classifier (meta-classifier)* (Wolpert 1992). The underlying idea is to learn whether training data have been properly learned.

SELF IMPLEMENTED ENSEMBLE APPROACH



```
class Ensemble:
    def __init__(self , data , classes):
        self.data = data
        self.classes = classes
        self.NNModel = self.createNNModel()
        self.NBClassifier = self.createNBModel()
        self.SVCLinear = self.createSVMModel('linear')
        self.DT = self.createDTModel()
        self.RF = self.createRFModel(100)
```

TRAINING THE MODELS INSIDE ENSEMBLE CLASS



```
ensemble = Ensemble(all_features , all_classes)
ensemble.trainModels()
```



```
Training Random Forest
RF Score : 0.7692307692307693
```

```
Training Support Vector Machine
SVM Score : 0.7692307692307693
```

```
Training Naive Bayes
Naive Bayes Score : 0.75
```

```
Training Neural Network
Epoch 1/20
622/622 [=====] - 2s 2ms/step - loss: 0.7912 - accuracy: 0.6785 - val_loss: 0.6172 - val_accuracy: 0.7837
Epoch 2/20
```

Accuracy of The ENSEMBLE APPROACH

Prediction is made after taking linear combination of predictions and accepting if $>70\%$ of models agree



```
ensemble.accuracy()
```

```
Accuracy: 0.7590361445783133
```

Machine Learning Assignment

Ujjwal Sharma - BT18CSE021