Machine Learning Report

Ujjwal Sharma - BT18CSE021

Chinmay Hattewar - BT18CSE055

Problem - Classification Problem

Predict whether a mammogram mass is benign or malignant

Dataset Used - Mammographic Mass Data

Url : https://archive.ics.uci.edu/ml/datasets/Mammographic+Mass)

1. Title: Mammographic Mass Data

2. Sources:

(a) Original owners of database:
Prof. Dr. R¸diger Schulz-Wendtland
Institute of Radiology, Gynaecological Radiology, University Erlangen-Nuremberg
Universit‰tsstrafle 21-23
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

(c) Date received: October 2007

3. Past Usage:

M. Elter, R. Schulz-Wendtland and T. Wittenberg (2007)

The prediction of breast cancer biopsy outcomes using two CAD approaches that both emphasize an intelligible decision process.

Medical Physics 34(11), pp. 4164-4172

4. 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.

5. Number of Instances: 961

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

7. 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)

8. Missing Attribute Values: Yes
   - BI-RADS assessment: 2
   - Age: 5
   - Shape: 31
   - Margin: 48
   - Density: 76
   - Severity: 0

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

Model Used

- DECISION TREES
- RANDOM FORESTS
- K-MEANS AND KNN
- SUPPORT VECTOR MACHINE
- NAIVE BAYES
- ARTIFICIAL NEURAL NETWORK

Decision Tree

A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements

Advantages

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

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.

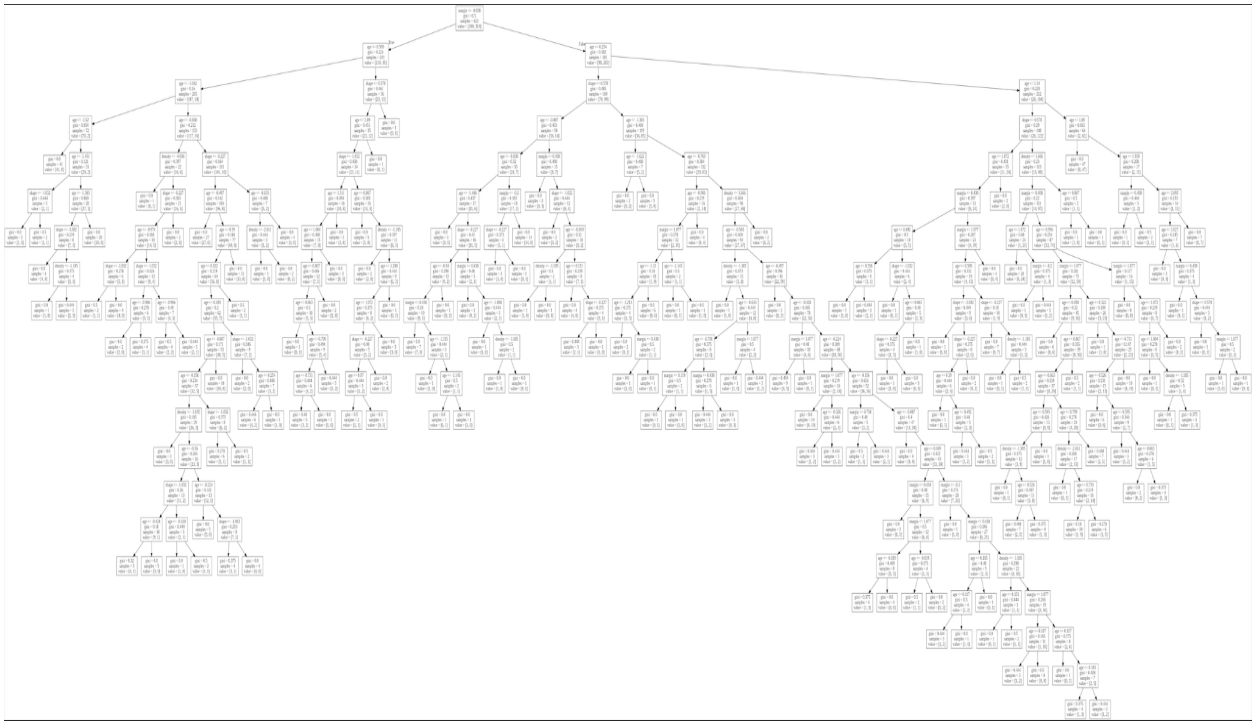
3. Unstable: Adding a new data point can lead to re-generation of the overall tree and all nodes need to be recalculated and recreated.
4. Affected by noise: Little bit of noise can make it unstable which leads to wrong predictions.
5. Not suitable for large datasets: If data size is large, then one single tree may grow complex and lead to overfitting. So in this case, we should use Random Forest instead of a single

Decision Tree.

API Used : sklearn

```
from sklearn.tree import DecisionTreeClassifier
decisionTreeClassifier = DecisionTreeClassifier(random_state=1)
# Training the classifier on the training set
decisionTreeClassifier.fit(training_inputs, training_classes)
decisionTreeClassifier.score(testing_inputs, testing_classes)
0.7355769230769231
```

Final Tree Build



```
from sklearn.model_selection import cross_val_score

decisionTreeClassifierKCross = DecisionTreeClassifier(random_state=1)
crossValidationScores = cross_val_score(decisionTreeClassifierKCross,
all_features_scaled, all_classes, cv=10)
crossValidationScores.mean()
0.7373493975903613
```

Also implemented ID3 algorithm

Dataset used for ID3 is play tennis (https://www.kaggle.com/fredericobreno/play-tennis)

File Name = ML Assignment ID3.ipynb
Inside folder - ML Algorithms Self Implemented/Decision Trees

```
##Making the classifier
###parameter1 : input Data -> testing Data
###parameter2 : label -> output class Name

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
```

I have also specified examples inside the notebook itself One such example is when days are also involved hence id3 algo giving the tree of length 1 but will have no predictive capability

```
treeDay = ID3(trainDataDay, 'play')
treeDay
{'day': {'D1': 'No',
  'D11': 'Yes',
  'D14': 'No',
  'D2': 'No',
  'D3': 'Yes',
  'D4': 'Yes',
  'D7': 'Yes',
  'D8': 'No',
  'D9': 'Yes'}}
accuracyDay = evaluate(treeDay , testDataDay , 'play')
print("Accuracy of Decision Tree : " , accuracyDay)
Accuracy of Decision Tree :  0.0
```

File Nam

Random Forest

Random forests or **random decision forests** are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned.[1][2] Random decision forests correct for decision trees' habit of overfitting to their training set.[3]:587–588 Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance.[4][5]

```
from sklearn.ensemble import RandomForestClassifier

randomForestClassifier100 = RandomForestClassifier(n_estimators=100)

randomForestClassifier100.fit(training_inputs, training_classes)
randomForestClassifier100.score(testing_inputs , testing_classes)
0.7692307692307693

randomForestClassifier200 = RandomForestClassifier(n_estimators=200)

randomForestClassifier200.fit(training_inputs, training_classes)
randomForestClassifier200.score(testing_inputs , testing_classes)
0.7596153846153846


Also done k cross validation
```

Advantages

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

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

***k*-means clustering** is a method of vector quantization, originally from signal processing, that aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. *k*-means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances. For instance, better Euclidean solutions can be found using k-medians and k-medoids.

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

kmeans = KMeans(n_clusters=2)

scaledFeatures = scale(all_features)

kmeans.fit(scaledFeatures)
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

Advantages 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 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.

K Nearest Neighbours

In statistics, the ***k*-nearest neighbors algorithm** (***k*-NN**) is a non-parametric classification method first developed by Evelyn Fix and Joseph Hodges in 1951,[1] and later expanded by Thomas Cover.[2] It is used for classification and regression. In both cases, the input consists of the *k* closest training examples in a data set. The output depends on whether *k*-NN is used for classification or regression:

This is self Implemented

fileName = ML_Assignment_KNN.ipynb
Inside Folder - ML Algorithms Self Implemented/KNN

```
trainingSet = []
testSet = []
split = 0.67
```

```
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 = getAcuuracy(testSet, predictions)
print("Accuracy: " + repr(accuracy) + " %")
Accuracy: 58.741
```

For k = 5
Accuracy - 61.11

For k = 7
Accuracy : 60.70

Advantages OF KNN

1. K-NN is pretty intuitive and simple: K-NN algorithm is very simple to understand and equally easy to implement. To classify the new data point K-NN algorithm reads through whole dataset to find out K nearest neighbors.

2. K-NN has no assumptions: K-NN is a non-parametric algorithm which means there are assumptions to be met to implement K-NN. Parametric models like linear regression has lots of assumptions to be met by data before it can be implemented which is not the case with K-NN.

3. No Training Step: K-NN does not explicitly build any model, it simply tags the new data entry based learning from historical data. New data entry would be tagged with majority class in the nearest neighbor.

4. It constantly evolves: Given it's an instance-based learning; k-NN is a memory-based approach. The classifier immediately adapts as we collect new training data. It allows

the algorithm to respond quickly to changes in the input during real-time use.

5. Very easy to implement for multi-class problem: Most of the classifier algorithms are easy to implement for binary problems and needs effort to implement for multi class whereas K-NN adjust to multi class without any extra efforts.
6. Can be used both for Classification and Regression: One of the biggest advantages of K-NN is that K-NN can be used both for classification and regression problems.

7. One Hyper Parameter: K-NN might take some time while selecting the first hyper parameter but after that rest of the parameters are aligned to it.

8. Variety of distance criteria to be choose from: K-NN algorithm gives user the flexibility to choose distance while building K-NN model.
1. Euclidean Distance
2. Hamming Distance
3. Manhattan Distance
4. Minkowski Distance

Disadvantages Of KNN

1. K-NN slow algorithm: K-NN might be very easy to implement but as dataset grows efficiency or speed of algorithm declines very fast.

2. Curse of Dimensionality: KNN works well with small number of input variables but as the numbers of variables grow K-NN algorithm struggles to predict the output of new data point.

3. K-NN needs homogeneous features: If you decide to build k-NN using a common distance, like Euclidean or Manhattan distances, it is completely necessary that features have the same scale, since absolute differences in features weight the same, i.e., a given distance in feature 1 must means the same for feature 2.

4. Optimal number of neighbors: One of the biggest issues with K-NN is to choose the optimal number of neighbors to be consider while classifying the new data entry.

5. Imbalanced data causes problems: k-NN doesn't perform well on imbalanced data. If we consider two classes, A and B, and the majority of the training data is labeled as A, then the model will ultimately give a lot of preference to A. This might result in getting the less common class B wrongly classified.

6. Outlier sensitivity: K-NN algorithm is very sensitive to outliers as it simply chose the neighbors based on distance criteria.

7. Missing Value treatment: K-NN inherently has no capability of dealing with missing

value problem.

Support Vector Machines

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane

Advantages

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

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.

API USED - SKLEARN

```
from sklearn import svm

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

svc.fit(training_inputs, training_classes)

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

svcKCross = svm.SVC(kernel=kernal, 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
```

- **For RBF KERNEL**

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

- **For Sigmoid Kernel**

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

- **For Polynomial Kernel**

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

Naive Bayes

In statistics, **naive Bayes classifiers** are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features (see Bayes classifier). They are among the simplest Bayesian network models,[1] but coupled with kernel density estimation, they can achieve higher accuracy levels.[2][3]

Naïve Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression,[4]:718 which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

In the statistics and computer science literature, naive Bayes models are known under a variety of names, including **simple Bayes** and **independence Bayes**.[5

Advantages

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

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**.

API USED - SKLEARN

```
import numpy
from sklearn.model_selection import train_test_split
scaler = preprocessing.MinMaxScaler()
#scaling the data b/w minimum and maximum
all_featuresMinMax = scaler.fit_transform(all_features)
numpy.random.seed(1234)
(training_inputsMinMax,
testing_inputsMinMax,
training_classesMinMax,
testing_classesMinMax) = train_test_split(all_featuresMinMax, all_classes,
train_size=0.75, random_state=1)

from sklearn.naive_bayes import MultinomialNB
naiveBayesClassifier = MultinomialNB()

naiveBayesClassifier.fit(training_inputsMinMax , training_classesMinMax)

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
```

Artificial Neural Networks

Artificial neural networks (**ANNs**), usually simply called **neural networks** (**NNs**), are computing systems inspired by the biological neural networks that constitute animal brains.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives a signal then processes it and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called *edges*. Neurons and edges typically have a *weight* that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

Advantages

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

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.

4. 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.
5. 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.

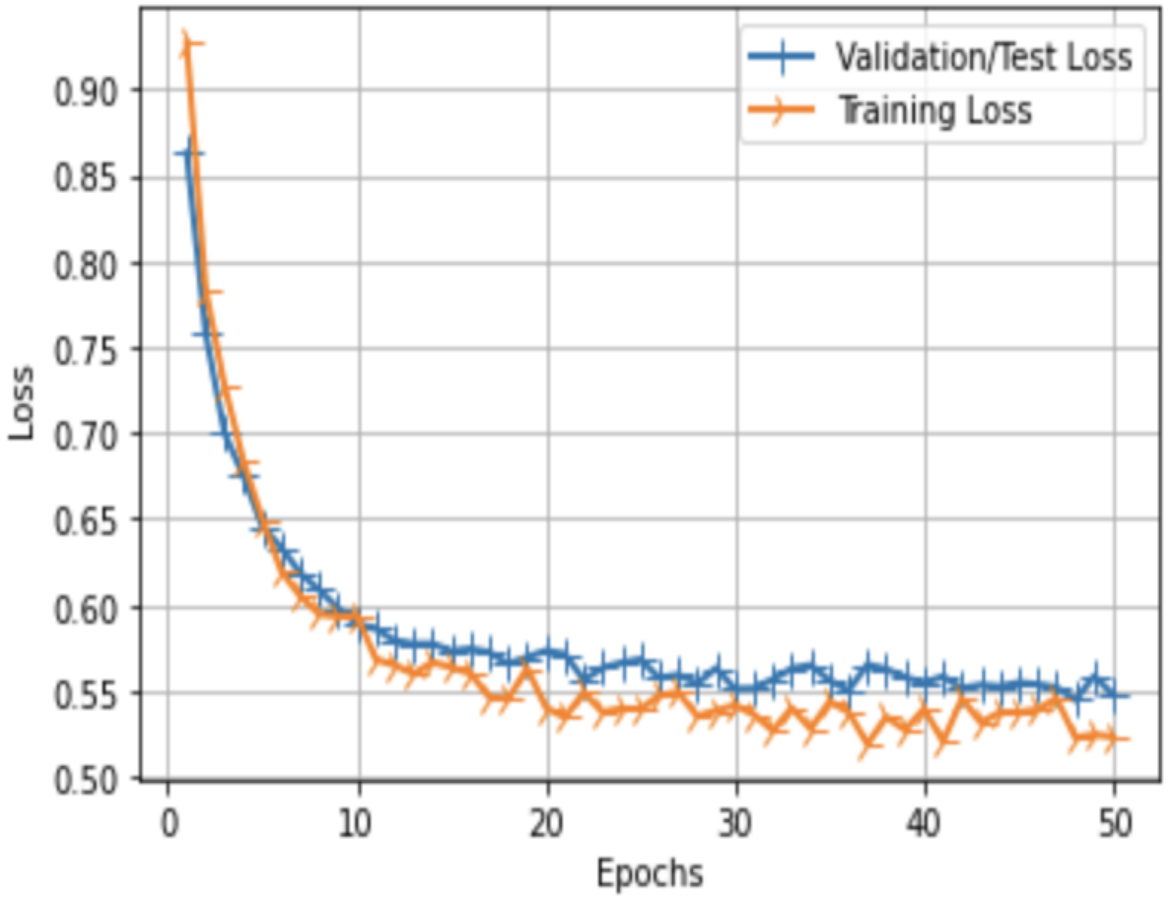
API USED - KERAS TENSORFLOW

Added Optimizer , Dropout , regularisation (l2) to avoid overfitting

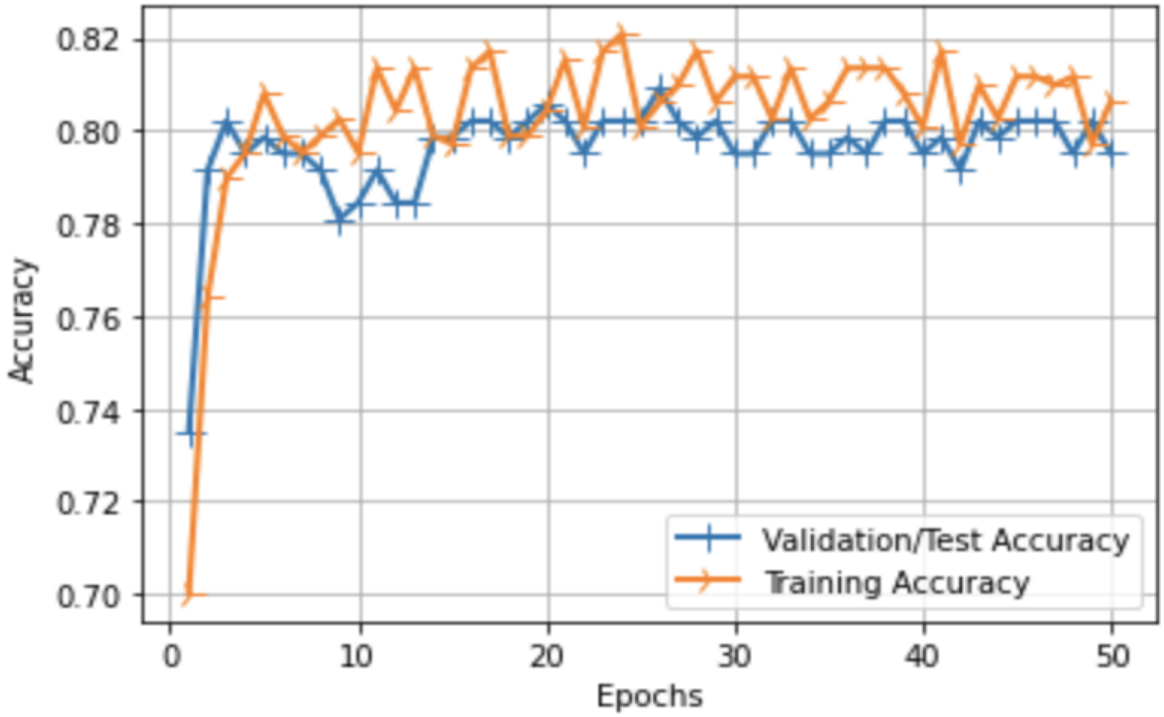
```
import keras
from keras.models import Sequential
from keras.layers import Dense , Activation , Dropout
import tensorflow


inputSize = training_inputs[0].shape

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'])

print(model.summary())
```

GRAPH OF LOSS VS EPOCHS
50 Epochs
batchSize = 4
Optimizer = RMSProp()



GRAPH OF LOSS VS EPOCHS
50 Epochs
batchSize = 4
Optimizer = RMSProp()



accuracy: 0.8062

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())
```

```
Average Accuracy with k = 10 is :  0.8072289109230042
```

Self Implemented Artificial Neural Network

fileName = ML_Assignment_ANN.ipynb
Inside Folder - ML Algorithms Self Implemented/Artificial Neural Network

Created a class Linear Layer
Then used this class inside the claa Neural Network

```
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
```

```
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)
```

..

I have also implemented Backpropagation Algorithm upto 3 layers and will generalize in the future

Currently It only supports upto 3 layers and internal parameters can be changed
I am doing stochastic gradient descent (i.e backprop after every example)

```
# This is the main function from which neural network will learn
   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()
```

Accuracy Of Self Implemented NN
10 Epochs
Learning Rate = 0.1
Accuracy = 77.866

Accuracy Of Self Implemented NN
10 Epochs
Learning Rate = 0.01
Accuracy = 57.21
Here the learning rate became too small so learning requires more epochs to converge

Accuracy Of Self Implemented NN
10 Epochs
Learning Rate = 0.3
Accuracy = 77.40

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 -th vector is the direction of a line that best fits the data while being orthogonal to the first 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.

PCA is used in exploratory data analysis and for making predictive models. It is commonly used for dimensionality reduction by projecting each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible. The first principal component can equivalently be defined as a direction that maximizes the variance of the projected data. The -th principal component can be taken as a direction orthogonal to the first principal components that maximizes the variance of the projected data.

From either objective, it can be shown that the principal components are eigenvectors of the data's covariance matrix. Thus, the principal components are often computed by eigendecomposition of the data covariance matrix or singular value decomposition of the data matrix. PCA is the simplest of the true eigenvector-based multivariate analyses and is closely related to factor analysis. Factor analysis typically incorporates more domain specific assumptions about the underlying structure and solves eigenvectors of a slightly different matrix. PCA is also related to canonical correlation analysis (CCA). CCA defines coordinate systems that optimally describe the cross-covariance between two datasets while PCA defines a new orthogonal coordinate system that optimally describes variance in a single dataset.[1][2][3][4] Robust and L1-norm-based variants of standard PCA have also been proposed.[5][6][4]

Simple Definition

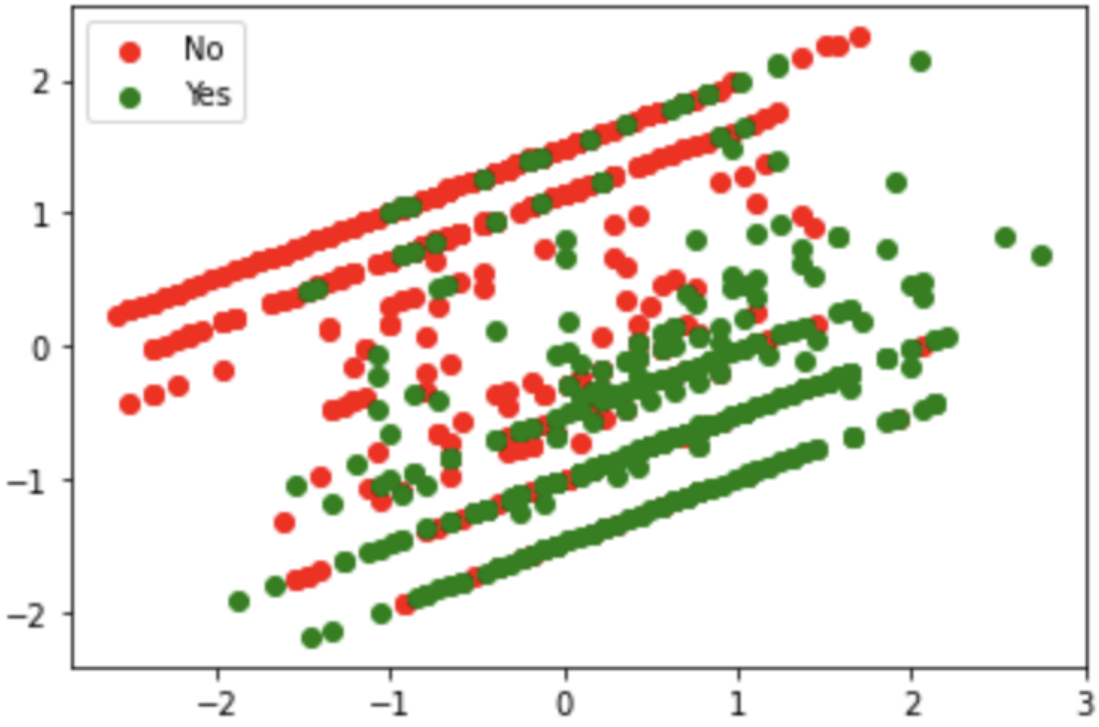
> PCA is a dimensionality reduction technique; it lets you distill multi-dimensional data down to fewer dimensions, selecting new dimensions that preserve variance in the data as best it can.

API USED- SKLEARN

```
from sklearn.decomposition import PCA
```

2 Components

```
from sklearn.decomposition import PCA
pca2 = PCA(n_components=2, whiten=True).fit(all_features)
allFeaturesPCA2 = pca2.transform(all_features)

lets see how much varience we preserved
print(pca2.explained_variance_ratio_)
print(sum(pca2.explained_variance_ratio_))
[0.98424958 0.01295367]
0.9972032568350249
```



KMEANS ON 2 Components

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

SVM On 2 Components

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

3 Components

```
pca3 = PCA(n_components=3, whiten=True).fit(all_features)
allFeaturesPCA3 = pca3.transform(all_features)
```

```
print(pca3.explained_variance_ratio_)
print(sum(pca3.explained_variance_ratio_))
[0.98424958 0.01295367 0.00224542]
0.9994486808625002
```

KMEANS ON 2 Components

```
Accuracy of K-means is :  0.7710843373493976
```

SVM On 3 Components

```
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.

Commonly used ensemble learning algorithms

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. That is, different training data subsets are randomly drawn – with replacement – from the entire training dataset. Each training data subset is used to train a different classifier of the same type. Individual classifiers are then combined by taking a simple majority vote of their decisions. For any given instance, the class chosen by most number of classifiers is the ensemble decision. Since the training datasets may overlap substantially, additional measures can be used to increase diversity, such as using a subset of the training data for training each classifier, or using relatively weak classifiers (such as decision stumps).

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. In essence, each iteration of boosting creates three weak classifiers: the first classifier C_1 is trained with a random subset of the available training data. The training data subset for the second classifier C_2 is chosen as the most informative subset, given C_1. Specifically, C_2 is trained on a training data only half of which is correctly classified by C_1, and the other half is misclassified. The third classifier C_3 is trained with instances on which C_1 and C_2 disagree. The three classifiers are combined through a three-way majority vote.

AdaBoost

Arguably the best known of all ensemble-based algorithms, **AdaBoost** (Adaptive Boosting) extends boosting to multi-class and regression problems (Freund 2001). AdaBoost has many variations, such as AdaBoost.M1 for classification problems where each classifier can attain a weighted error of no more than 1/2 , AdaBoost.M2 for those weak classifiers that cannot achieve this error maximum (particularly for problems with large number of classes, where achieving an error of less than 1/2 becomes increasingly difficult), AdaBoost.R (for regression problems), among many others.

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. For example, if a particular classifier incorrectly learned a certain region of the feature space, and hence consistently misclassifies instances coming from that region, then the Tier 2 classifier may be able to learn this behavior, and along with the learned behaviors of other classifiers, it can correct such improper training. Cross validation type selection is typically used for training the Tier 1 classifiers: the entire training dataset is divided into T blocks, and each Tier-1 classifier is first trained on (a different set of) T-1 blocks of the training data. Each classifier is then evaluated on the Tth (pseudo-test) block, not seen during training.

I Have tried implementing 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)
```

```
 def predict(self , feature_in):
   a = 5
   norm = np.linalg.norm(feature_in)

   feature = np.array([feature_in/norm])
   feature_inNB = np.array([feature_in])
   nn = self.NNModel.predict(feature)[0]
   # print(nn)
   if nn<0.8:
     nn = 0
   else:
     nn = 1
   nb = self.NBClassifier.predict(feature_inNB)[0]
   svc = self.SVCLinear.predict(feature)[0]
   dt = self.DT.predict(feature)[0]
   rf = self.RF.predict(feature)[0]
   # print([nn , nb , svc , dt , rf])
   tval = (nn+nb+svc+dt+rf)/a
   # print(tval)
   if tval>=0.7:
     return 1
   return 0

 def accuracy(self):

   (training_inputs,
   testing_inputs,
   training_classes,
   testing_classes) = train_test_split(self.data, self.classes,
train_size=0.5, random_state=1)
   from sklearn import metrics
   predictions = []
   for row in testing_inputs:
     predictions.append(self.predict(row))
   predictions = np.array(predictions)

   print("Accuracy:",metrics.accuracy_score(testing_classes, predictions))
```

Some of the code is omitted intentionally . To see the whole code open the file - ML Project using APIs.ipynb

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

Training Decision Tree
DT Score :  0.7355769230769231


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: 0.7571 -
accuracy: 0.7508 - val_loss: 0.6382 - val_accuracy: 0.7596
Epoch 2/20 …..



ensemble.accuracy()
75.58 %

NOTE:
```

Prediction is made after taking linear combination of predictions and accepting if $>70\%$ of models agree I can modify this combination for giving me good result

UJJWAL SHARMA - BT18CSE021

**********END**********