Unit 5

Machine Learning?

Machine Learning? → programs that automatically adjust their performance in accordance with their exposure to information in data.

→ models are parametrized, with tunable parameters that adjust themselves on the basis of performance criteria.

→ ML is a subset of AI

Types of Machine Learning?

1. **Supervised Learning**: → algorithms that learn from a training set of labeled examples, to generalize to all possible inputs
   → eg. logistic regression, decision trees

2. **Unsupervised Learning?**: → algorithms that learn from a training set of unlabeled data
   → data is explored based on statistical, geometric or similarity criterion
   → eg. k-means clustering, kernel density estimation

3. **Reinforcement Learning?**: → algorithms that learn via criticism from the quality of the solution
   → improved solutions are achieved by iterative exploration

Nomenclature: Input data is in the form of numpy arrays
The size of the array is [n_samples, n_features]
↓ vertical ↓ horizontal

n-samples : an item to process

n-features : distinct traits to describe items in a quantitative manner

Columns : features, attributes, regressors, covariates, independent variables

Raws : instances, samples

Label : outcome, response or dependent variable.

* To find the no. of samples and features

if x is the feature matrix w/ [n samples, n features]

sample = x.shape[0]

feature = x.shape[1]

Supervised Learning : aims to find a mapping fn. from the input variable to the output variable.

Classification Algorithms : Classification is regarded as the problem of finding a function $h(x) : R^d \longrightarrow K$, that maps an input space Rd into a discrete set of k target output of classes $K = \{1, 2 \dots k\}$

Types of classification : (i) binary classifiers
(ii) multiclass classifiers

k-NN (K-Nearest Neighbour) Algorithm

→ The k-NN algorithm assumes the similarity between the available cases and the new cases, and puts the new case into a category which is most similar to the available categories

→ k-NN used for both regression & classification, but mostly for classification.

→ a kind of non-parametric algorithm, does not make any assumptions on the underlying data.

→ called a lazy-learning algorithm, because it does not learn from the training set immediately, but rather stores the dataset and ~~stores~~ uses it at the time of classification

Algorithm : 1. Select the no. K of the neighbours

2. Calculate the Euclidean distance of K neighbours

3. Take the K-nearest neighbours as per the calculated Euclidean distance.

4. Among these k neighbours, count the no. of datapoints in each category.

5. Assign new data points to that category, ~~wi~~ for which the no. of neighbors is maximum.

Code: (running the classifier on the entire dataset)

```
# opening & reading the dataset

import pickle
f = open ('          ', 'rb')
(x,y) = pickle.load (f, encoding = ' ')
   └→ x = feature matrix
      y = labelled vectors

# developing the classifier.

from sklearn import neighbours.          → create an instance of knnclassifier
k = ___
knn = neighbors.KNeighborsClassifier (k)
```

```
#train the classifier
knn.fit(x,y)

#predict the result
pred = knn.predict(x)
```

*Training and Testing Data

Training Data: To learn the instance of the model from a model class

Testing data: To assess performance at the end of the training process

→ The data is split into training and testing data. The split for training & testing is usually 80-20 or 70-30.

Splitting the data set

① Using in-built sklearn functions

```
from sklearn.model_selection import test_train_split

prc=0.3     # for a 70-30 split

X_train, Y_train, X_test, Y_test = train_test_split
                                   (x,y, test_size=prc)
```

② Using manual randomization ~~randes~~

```
import numpy as np
perm = np.random.permutation(y.size)
prc = 0.3
split_point = int(np.ceil(y.shape[0] * PRC))
X_train = x[perm[:split_point].ravel(),:]
Y_train = y[perm[:split_point].ravel()]
X_test = x[perm[split_point:].ravel(),:]
Y_test = y[perm[split_point:].ravel()]
```

* # Running the model on the testing and training (3)

data

training data

```
from sklearn import neighbors

knn= neighbors. KNeighborsclassifier( n_neighbors = 5 )

knn. fit( X_train, Y_train)

train= knn. predict (X_train )
```

testing data

```
knn. fit ( X_test , Y_test )

pred= knn. predict ( X_test)
```

* # To test the accuracy of the model

(1) To find the % accuracy

```
knn. score ( X_test ; Y_test )
```

(2) To find the classification accuracy for the training and testing data, as well as the confusion matrix

```
From sklearn import metrics
```

on training data

```
print ( metrics. acauracy- score ( train, y_train)

print ( metric. confusion- matrix ( y_train , train)
```

on testing data

```
print ( metrics. accuracy - score ( pred , y_test ))
print ( metric s. confusion - matrix ( pred , y_test ))
```

Performance Metrics

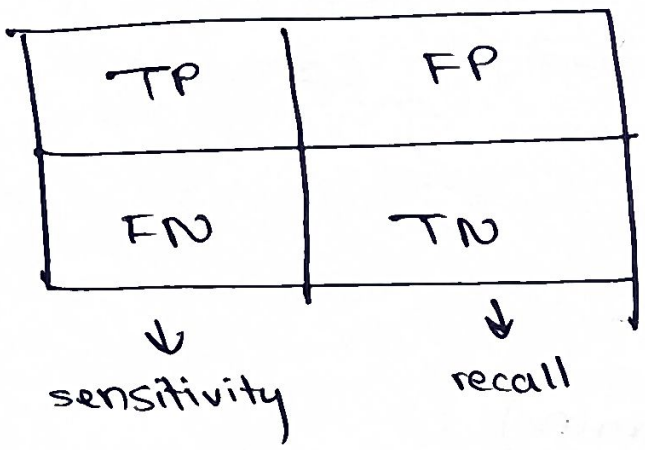
(1) Confusion matrix: The confusion matrix considers the outcomes of the classifier and the actual ground truth. The four possible cases are:

(a) True Positives (TP): classifier predicts as tve and it really is tve

(b) False Positives (FP): classifier predicts as tve, but in fact is actually -ve

(c) True Negatives (TN): classifier predicts as -ve, it really is -ve

(d) False Negatives (FN): classifier predicts as -ve, but is in fact +ve.



(e) Accuray $= \frac{TP + TN}{TP + TN + FP + FN}$

(3) Sensitivity or recall : $\frac{TP}{TP+FN}$

(4) Specificity : $\frac{TN}{TN+FP}$

} column-wise

(5) precision or +ve predictive value : $\frac{TP}{TP+FP}$

(6) negative predict value = $\frac{TN}{TN+FN}$

} row-wise

Generation of the confusion matrix using a formula

```
pred = knn.predict(x)

TP = np.sum(np.logical_and(pred == -1, y == -1))

TN = np.sum(np.logical_and(pred == 1, y == 1))

FP = np.sum(np.logical_and(pred == -1, y == 1))

FN = np.sum(np.logical_and(pred == 1, y == -1))
```

To generate a piechart showing the outcomes

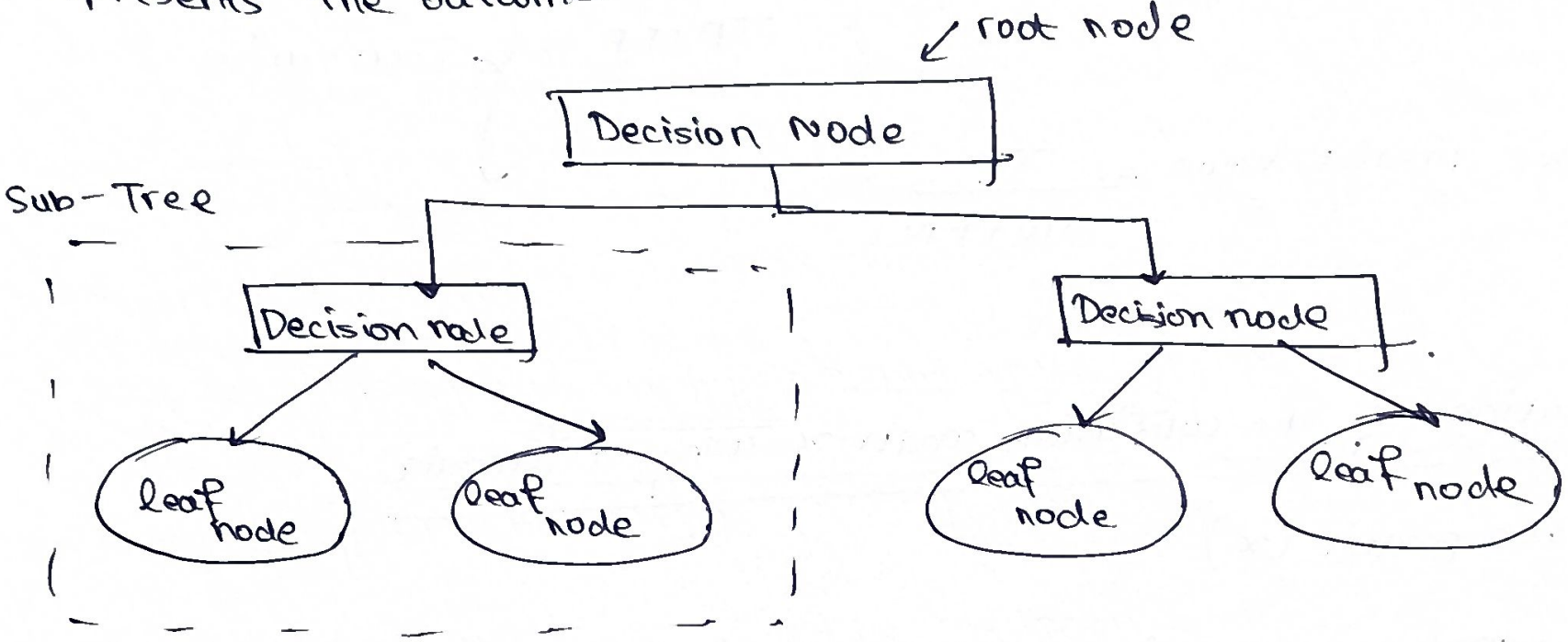
```
import numpy as np
import matplotlib.pyplot as plt
plt.pie(np.c_[[np.sum(np.where(y==1,1,0)), np.sum(np.where(y==-1,1,0))]][0],
        labels=['  ', '  '],
        colors=['g', 'r'])
plt.savefig("  .png")
```

np.c_ → concatenates rows of a nested array

np.where → condition, if true, if false
do do

Decision Tree Algorithm

→ A decision tree is a flowchart-like tree structure, where internal nodes represent the features of a dataset, branches represent decision rules, and each leaf node represents the outcome.



→ It is a kind of greedy algorithm, and the tree is constructed in a top-down recursive divide and conquer manner.

→ At the start, all training examples are at the root.

→ Attributes are categorical, if they are continuous, then they are discretized in advance.

→ Examples are partitioned recursively based on selected attributes, test attributes are selected on the basis of a heuristic or statistical measure. (say, for eg. information gain).

Algorithm

1. Select the best attribute using Attribute Selection Measures to split the records.

2. Make that attribute a decision node and break the dataset into smaller subsets.

3. Repeat this process recursively for each node, until one of the following conditions match:

(i) all of the tuples belong to the same attribute value

(ii) there are no more remaining attributes

(iii) there are no more samples

Attribute Selection Measure : Information Gain

→ used to decide the best feature that gives maximum information about a class.

Expected information : → the info needed to classify a tuple in D.

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

Information (after splitting D into v partitions)

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times I(D_j)$$

Information gain: $Gain(A) = Info(D) - Info_{(A)}(D)$

Code

```
from sklearn import metrics
from sklearn import tree
prc=0.7
x_train, x_test, y_train, y_test = train_test_split( x,y, test_size
                                                    =prc )
dt = tree.DecisionTreeClassifier( )
dt.fit ( x_train, y_train)
```

```
pred = dt.predict(x_test)
print(metrics.accuracy_score(pred, y_test))
```

CART

→ CART = Classification and Regression Trees

→ a subset of the decision tree algorithm,

→ The attribute selection is done using the Gini index.

$$\text{Gini Impurity} = 1 - \text{Gini} = 1 - \sum_{i=1}^{n} pi^2$$

Algorithm

→ The gini impurity is calculated for each node.

→ The weighted sum of the gini impurities for every child node is calculated.

→ The node with the lowest gini impurity is where the split happens.

Code

```
# import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import tree
x_train, y_train, x_test, y_test = train_test_split
```

Regression Analysis

Regression is used to make predictions, on the basis of a combination of one or more independent variables

(1) Simple Linear Regression

→ Simple linear regression consists of n samples of a single variable $x \in R$, and describes the relationship between the variable and response with the model:

$$y = a_0 + a_1 x$$

↓

intercept / constant term

The parameters (a's) are chosen using the ordinary least square method to minimize the square of the distance between the actual & predicted values

$$ORS = (y_{pred} - y_{obs})^2$$

Code:

```
from sklearn.linear_model import LinearRegression
from sklearn import metrics
prc = 0.3
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
                                                    = prc)
model = LinearRegression()
model.fit(x_train, y_train)
y_obs_train = model.predict(x_train)
y_obs_test = model.predict(x_test)
```

```
#evaluation
print('MSE :', metrics.mean_squared_error(y_obs,y_test,y)
print('R^2:', metrics.r2_score(y_obs, y_test)
```

(ii) Logistic Regression

→ a kind of binary classification model

→ used to predict the outcome of a categorical dependent variable based on one or more predictor variables.

The Logistic function is: $f(x) = \frac{1}{1+e^{-\lambda x}}$

Code:

```
# partitioning the data into the dependent & independent variables
y = df['outcome']          #boolean
x = df.drop['outcomes', axis=1]    # has all the other columns

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

x_train, x_test, y_train, y_test = train_test_split(x,y, size=0.3)

reg_model = LogisticRegression()
reg_model.fit(x_train, y_train)
pred_train = reg_model.predict(x_train)
pred_test = reg_model.predict(x_test)

## Evaluation
(i) score
print(reg_model.score(x_train, y_train)
print(reg_model.score(
```

Evaluation

```
print (metrics. acauracy_ score( pred_train, y_train)
print (metrics acauracy_score( pred_test , y_test )
print (metric . confusion_matrix ( pred_test , y_test))
```

Unsupervised Learning

→ A set of algorithms which learn from a traing set of unlabeled or unannotated examples.

→ The features of the inputs are categorized according to some geometric or statistical criteria.

Clustering : aims to partition the set of examples into groups

- Examples within a cluster are similar
- Examples in different clusters are difficult

Inputs : → in a similarity-based clustering

input = n x n dissimilarity matrix (distance matrix)

→ in feature based clustering

input = n x D feature matrix

Similarities and distance

The most widely used distance metric is the Minkowski distance:

$$d(a,b) = \left(\sum_{i=1}^{d} |a_i - b_i|^p \right)^{1/p}$$

Other instantiations are:

$p = 2 \Rightarrow$ euclidean distance

$p = 1 \Rightarrow$ Manhattan distance

$p = \infty \Rightarrow$ max distance

Clustering Techniques

(i) partitional algorithms : start w/ a random partition & refine iteratively

(ii) hierarchical algorithms: agglomerative (bottom up) / top down

Partitional Algorithms

(i) Hard partitional algorithms: K-means (assign a unique cluster value to each element in the feature space)

(ii) Soft partition algorithms: mixture of gaussians, which assigns a confidence / probability to each point in space.

K-Means Clustering

→ a kind of hard partitioning algorithm which has the goal of assigning each data point to a single cluster.

→ The algorithm divides a set of n samples X in k disjoint clusters, C_i, $i = 1, 2, 3 \ldots k$, each cluster defined by the mean of the samples in the cluster.

→ These means are called centroids.

→ The objective function of K-means clustering is the square of the Euclidean distance : $d(x, \mu_j) = ||x - \mu_j||^2$, which is also called the inertia or the within-cluster sum of squares (wcss)

Algorithm

1. Initialize the value K of desirable clusters
2. Initialize the cluster centers randomly
3. Decide the class memberships of the N data samples by assigning them to the nearest cluster centroid (the mean)
4. Re-estimate the K cluster centers, assuming that the memberships are correct.
5. If none of the N objects changed membership in the last iteration, exit. Otherwise, go to step 3.

Code! **Step1:** Find the value of K.

→ The kmeans algorithm's efficiency is highly dependent on the no. of clusters.

→ The elbow method can be used to calculate the ideal no. of clusters within the dataset.

→ This methode uses the (Within Cluster Sum of Squares (WCSS) to define total variations.

```
from sklearn.cluster import KMeans
wcss = []
for i in range(1,10):
    kmeans = KMeans(n_clusters = i, init = 'k-means++',
                    randomstate = )
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)
```

```
kmeans = KMeans ( n_clusters = ,  random. stat = 0)

clusters = kmeans . fit_predict (  )

from sklearn . metrics  import accuracy.score

accuracy_score ( clusters, [illegible] )
```