Western Engineering

ECE 9013A (M.ENG)

Programming for Engineers

Project Report

PREDICTING KIDNEY DISEASE USING MACHINE LEARNING ALGORITHMS

Group 4

Name	Student Number
Xiangliang Yu	250796161
Uriev Ellapen	251059007
Tanveer Alam	251124105
Sathyanath Nandakumar	251085351

TABLE OF CONTENTS

INTRODUCTION

Chronic Kidney Disease aka CKD has become a global health problem. CKD is the gradual loss of kidney function in the human body. The function of kidney in a human body is to filter the waste and other excess fluids from the blood which are then released from the human body via urine. CKD is the measure of how much one's kidney has damaged. When CKD reaches an advanced stage, different fluids, electrolytes and wastes can build up and create imbalance in human body resulting in kidney failure.

Over two million people worldwide receive daily dialysis or require a kidney transplant. Thus, it is of great importance to detect CKD at an early stage and control it. As computers are getting better in understanding the large set of data due to breakthroughs in data mining, the concept of virtual physicians for patients is becoming popular.

In this project, three machine learning algorithms, namely KNN, SVM and Decision Tree, are used to train the computer program from the clinical dataset, collected by the physicians to gain insights about the kidney disease, and based on this training these algorithms predict whether the patient has CKD or not. Features on the clinical dataset are age, albumin, blood urea, serum creatinine, high blood pressure, and diabetic mellitus. These machine learning algorithms are run with different parameter settings and the comparison of all the algorithms are done based on the metrices Accuracy, Positive Predicted Value, Negative Predicted Value Sensitivity, and Specificity.

PROBLEM DEFINITION AND MOTIVATION

Problem Definition

The goal is to create a program that will predict whether the patient has kidney disease or not based on the training data using the features: age, albumin, blood urea, serum creatinine, high blood pressure, and diabetic mellitus. The output of the algorithms is also verified with the actual data. The tasks involved are the following:

1. Download and preprocess the clinical dataset
2. Train the classifier with KNN that can determine if the patient has kidney disease or not.
3. Train the classifier with SVM that can determine if the patient has kidney disease or not.
4. Train the classifier with Decision Tree that can determine if the patient has kidney disease or not.
5. Compare and contrast KNN, SVM, and Decision Tree.
6. Create a Graphical User Interface for the program

The result of the code will be GUI from which results of the three mentioned algorithms can be run and seen with just a click.

Motivation

The project is inspired by the data found in the National Kidney Foundation. The data states that annually more than one million people in 112 low income countries, die from kidney disease. It's better for people of a lower income to detect kidney disease at an early stage to allow for less expensive cures as opposed to late stage treatments such as dialysis or a kidney transplant. The program of this project will help them in doing so [3].

SUPPORT VECTOR MACHINES

INTRODUCTION

Support Vector Machines (SVM) is an established supervised classification algorithm implemented in Python for machine learning. In this project an SVM algorithm was used for binary classification to predict whether a patient will have chronic kidney disease depending on medical data. SVM constructed a hyperplane in multidimensional space to separate the two classes into chronic kidney disease (cdk) and not chronic kidney disease (notcdk). The main objective is to select a hyperplane with the maximum possible margin between support vectors using the given dataset.

To illustrate the concept, consider the case of linearly separable data in two dimensions. When the hyperplane is created two points are picked which are known as support vectors. The support vectors must be the closest points to the hyperplane and their distance from the hyperplane must be identical [1]. With this rule many hyperplanes can be created which is used to classify the medical data. SVM chooses the best hyperplane which has the greatest possible distance between the support vectors. The largest margin between support vectors and the hyperplane ensures the most accurate classification.

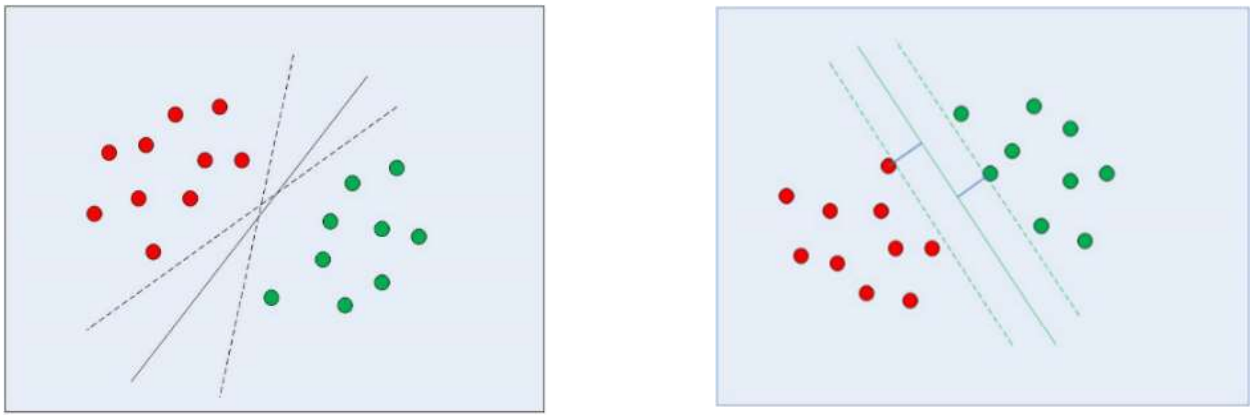


Fig 1. Multiple hyperplanes and hyperplane with support vector for linear separable data.

Not all problems can be solved using linear hyperplanes. In the case of non-linearly separable data SVM uses a kernel to transform the input space into a higher dimension. The kernel converts any non-separable problem to separable problems by increasing the dimension [1]. The purpose of the kernel is to plot the data in a higher dimension so that it may be divided using a hyperplane. The types of kernels available in Python are the linear, polynomial, radial basis function, and sigmoid kernel.

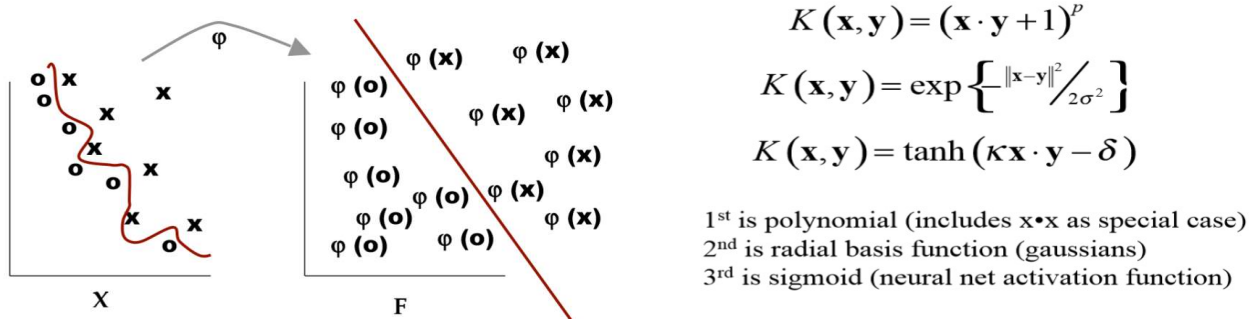


Fig 2. Python kernels applied to non-linearly separable dataset

INTRODUCTORY MATHEMATICS BEHIND SVM

The problem of finding the optimal hyperplane is an optimization problem that can be solved using Lagrange multipliers [2]. The aim of SVM is to orientate the hyperplane as far as possible from the closest support vector of both classes.

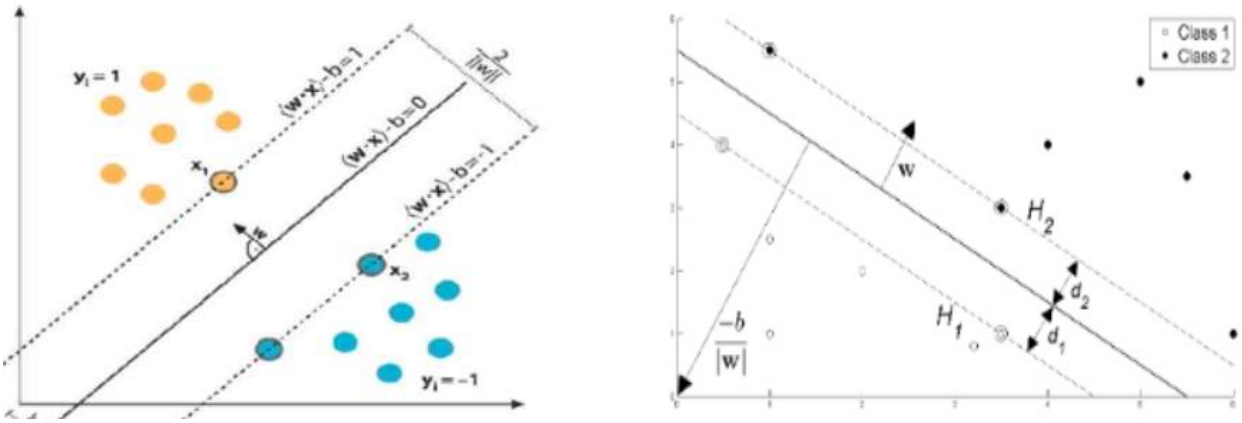


Fig 3.Mathematical interpretation of optimal hyperplane

From Fig 6, the SVM problem may be formulated as

$w.x_i + b \geq 1 \, for \, y_i = +1$

$w.x_i + b \leq 1 \, for \, y_i = -1$

Combining the above two equations can be written as

$y_i(w.x_i + b) - 1 \geq 0 \, for \, y_i = +1, -1$

The two hyperplanes H1 and H2 pass through the support vectors of +1 and -1 respectively. The distance between H1 hyperplane and the origin is $(-1 - b)/\| w \|$, the distance between H2 hyperplane and the origin is $(1 - b)/\| w \|$. The margin is given as the difference between these two distances [2].

$M = (1 - b)/\| w \| - (-1 - b)/\| w \|$

$M = 2/\| w \|$

M is twice the margin therefore the margin can be rewritten as $1/\| w \|$. The objective of SVM is to choose an optimal hyperplane that maximizes the margin which boils down to maximizing the $1/\| w \|$ term [2]. Solving SVM is a constrained optimization problem and Lagrange multipliers are used to solve it. For continued exposition of the mathematics behind SVM please refer to the references section of the report.

METHODOLOGY

1. DATA PREPROCESSING

The snapshot below shows the process_split function used on the kidney dataset.

```
def process_split(dt, ts):  # function for processing the values and splitting into training and testing sets
    dtc = list(dt.columns.values)
    label = preprocessing.LabelEncoder()
    for i in dtc:
        dt[str(i)] = label.fit_transform(list(dt[str(i)]))
    xc = dt.values[:, 0:6]
    yc = dt.values[:, 6]
    xt, xts, yt, yts = sklearn.model_selection.train_test_split(xc, yc, test_size=ts)
    return xt, xts, yt, yts
```

Description:

- The purpose of preprocessing is to take the dataset and encode them into integer values.
- The data is split into attributes and class output which is used for training and testing.
- Preprocessing is imported from sklearn in order to do this.
- The process_split(dt,ts) function can work with datasets which have an arbitrary number of columns.

2. SVM LINEAR AND RBF KERNEL MAIN FUNCTION CALL

For the SVM algorithm implemented in this project the linear and rbf kernels were used. The polynomial kernel was tested and has high accuracy but was not used due to the long processing time to execute the algorithm. The snapshots below show the kidneyfitsvm() and kidneyfitsvm_rbf() functions implemented on the kidney data set for linear and rbf kernels.

```
def kidneyfitsvm():
    mostaccurate = 0
    kdata = file.import_files2()
    for j in range(5):
        x_train, x_test, y_train, y_test = process_split(kdata, 0.2)
        fitsvm = svm.SVC(kernel="linear", C=2)
        fitsvm.fit(x_train, y_train)
        y_pred = fitsvm.predict(x_test)
        acc = metrics.accuracy_score(y_test, y_pred)
        if acc > mostaccurate:
            mostaccurate = acc
            with open("svmkidney.pickle", "wb") as f:
                pickle.dump(fitsvm, f)
    print("Accuracy:", mostaccurate, "\n")
    pickle_in = open("svmkidney.pickle", "rb")
    fitsvm = pickle.load(pickle_in)
    predicted = fitsvm.predict(x_test)
    names = ["ckd", "notcdk"]
    for x in range(len(predicted)):
        print("Predicted:", names[predicted[x]], " Data:", x_test[x], " Actual:", names[y_test[x]])
    return x_train, x_test, y_train, y_test, predicted
```

```
def kidneyfitsvm_rbf():
    mostaccurate = 0
    kdata = file.import_files2()
    for j in range(5):
        x_train, x_test, y_train, y_test = process_split(kdata, 0.2)
        fitsvm = svm.SVC(kernel="rbf", C=2, gamma='auto')
        fitsvm.fit(x_train, y_train)
        y_pred = fitsvm.predict(x_test)
        acc = metrics.accuracy_score(y_test, y_pred)
        if acc > mostaccurate:
            mostaccurate = acc
            with open("svmkidney.pickle", "wb") as f:
                pickle.dump(fitsvm, f)
    print("Accuracy:", mostaccurate, "\n")
    pickle_in = open("svmkidney.pickle", "rb")
    fitsvm = pickle.load(pickle_in)
    predicted = fitsvm.predict(x_test)
    names = ["ckd", "notcdk"]
    for x in range(len(predicted)):
        print("Predicted:", names[predicted[x]], " Data:", x_test[x], " Actual:", names[y_test[x]])
    return x_train, x_test, y_train, y_test, predicted
```

Description:

- Sklearn is used to import SVM and the kernel criterion is set as linear and rbf. The soft margin for both algorithms is set as 2.
- 80% of data used to train and the other 20% used to test.
- The models are created and training data fit onto it, the predicted values are compared to the correct values and the accuracy of both SVM models are established and printed to the screen.
- The for loop executes 5 times and stores the model with the highest accuracy using pickle which is imported.

3. PERFORMANCE OF THE SVM ALGORITHM

The two main functions that are used to establish the SVM algorithm performance are kidneyaccuracy(x,y) and getPerformance(TruePositive, FalsePositive, TrueNegative, FalseNegative).

```
def kidneyaccuracy(x, y):
    tp = 0
    tn = 0
    fp = 0
    fn = 0
    for j in range(len(y)):
        if x[j] == y[j] == 0:
            tp += 1
        if x[j] == y[j] == 1:
            tn += 1
        if x[j] == 0 and y[j] != x[j]:
            fp += 1
        if x[j] == 1 and y[j] != x[j]:
            fn += 1
    print("\n" + "Count of true positive=", tp, " Count of true negative=", tn)
    print("Count of false positive=", fp, " Count of false negative=", fn, "\n")
    return tp, tn, fp, fn
```

```
def getPerformance(TruePositive, FalsePositive, TrueNegative, FalseNegative):
    Accuracy = accuracy(TruePositive, FalsePositive, TrueNegative, FalseNegative)
    Sensitivity = sensitivity(TruePositive, FalseNegative)
    Specificity = specificity(TrueNegative, FalsePositive)
    ppv = PositivePredictedValue(TruePositive, FalsePositive)
    npv = NegativePredictedValue(TrueNegative, FalseNegative)
    print(' Accuracy =', round(Accuracy, 3))
    print(' Sensitivity =', round(Sensitivity, 3))
    print(' Specificity =', round(Specificity, 3))
    print(' Pos. Pred. Val. =', round(ppv, 3))
    print(' Neg. Pred. Val. =', round(npv, 3), "\n")
    return (Accuracy, Sensitivity, Specificity, ppv, npv)
```

Description:

- True Positive: predicted positive vs actual positive.
- True Negative: predicted negative vs actual negative.
- False Positive: predicted positive vs actual negative.
- False Negative: predicted negative vs actual positive
- Accuracy, sensitivity, specificity, positive predicted value, and negative predicted value is calculated.

4. OUTPUT:

SVM Linear Kernel Output Results

```
Accuracy = 0.985
Sensitivity = 0.974
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.964
```

SVM Radial Basis Function Kernel Output Results

```
Accuracy = 0.909
Sensitivity = 0.973
Specificity = 0.828
Pos. Pred. Val. = 0.878
Neg. Pred. Val. = 0.96
```

KNN (K – NEAREST NEIGHBOURS)

INTRODUCTION

K Nearest Neighbor or KNN is the simplest machine learning algorithm that labels object based on the nearest examples present on the training dataset in the feature space. The method simply stores the entire training dataset during the learning phase and assigns to each query a label represented by the majority label of its k-nearest neighbors in the training set. This method works is such a way that each sample should be classified like its surrounding samples. Therefore, a sample class is unknown then it could be predicted by considering the label of its nearest neighbors.

Given a sample that needs to be predicted with a training dataset, the distances between the unknown sample and all the data present in the training dataset can be computed. After calculating the distance, the distance with the smallest value corresponds to the sample in the training set closest to the unknown sample. Hence, the unknown sample may be labelled as the same as that of the label of the nearest neighbors. [4-9]

Figure shows the working of this algorithm at two values of k i.e. 1 and 4 for a set of examples divided into two classes. In figure 1(a), the unknown sample is labelled by using only one known nearest sample as the value of k is one, whereas in figure 1(b) closest four samples are considered for the classification of unknown. Three of them belong to the same class, whereas only one belongs to the other class. In both cases, the unknown sample is classified as belonging to the class on the left.

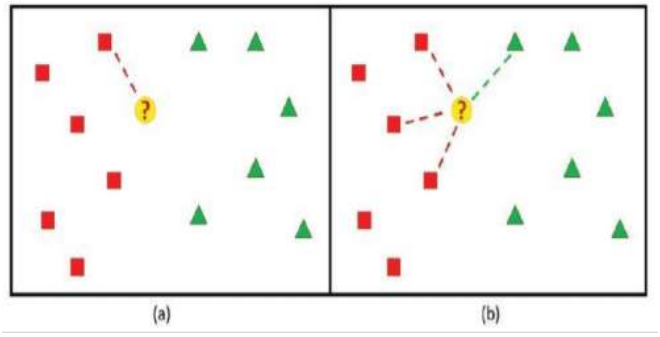


Figure 4

METHODOLOGY

1. DATA PREPROCESSING

Since the dataset consists of some of the features that are non-numeric, therefore these values need to be converted to numeric values for the calculation of distance between test data and the training dataset. For this purpose, linear_model preprocessing is imported from sklearn library and it is used to transform the data of each feature into numeric values

```
kidney = preprocessing.LabelEncoder()
age = kidney.fit_transform(list(data["Age"]))
al = kidney.fit_transform(list(data["Albumin (al)"]))
bu = kidney.fit_transform(list(data["Blood Urea (bu)"]))
sc = kidney.fit_transform(list(data["Serum Creatinine (sc)"]))
htn = kidney.fit_transform(list(data["High Blood Pressure (htn)"]))
dm = kidney.fit_transform(list(data["Diabetes Mellitus (dm)"]))
cls = kidney.fit_transform(list(data["class"]))
```

After the conversion, all the features are stored into variable x via list and the class is stored into variable y.

Now the preprocesses dataset is split into training and test data, and for this, split function is called.

```
def split(examples):
    randomSamples = random.sample(range(len(examples)), len(examples) // 5)
    train, test = [], []
    for i in range(len(examples)):
        if i in randomSamples:
            test.append(examples[i])
        else:
            train.append(examples[i])
    return train, test

x_train, x_test = split(x)
y_train, y_test = split(y)
```

2. FINDING THE K NEAREST NEIGHBOURS AND THEIR DISTANCES

For this purpose, the function KNearestNeighbour is used

```
def KNearestNeighbours(k=3):
    examples = buildKidneyExamples('Kidney_Data1.txt')
    train, test = split(examples)
    for sample in test:
        nearest, distances = findKNearest(sample, train, k=3)
    print("Object: ", nearest)
    print("Distance: ", distances, "\n")
    return None
```

This function will call function buildKidneyExample:

```
def buildKidneyExamples(fileName):
    data = getKidneyData(fileName)
    examples = []
    for i in range(len(data['age'])):
        p = Patient(data['age'][i], data['al'][i], data['bu'][i], data['sc'][i], data['htn'][i], data['dm'][i],
                    data['ckd'][i])
        examples.append(p)
    print('Finished processing', len(examples), 'patients\n')
    return examples
```

The above function will take the clinical dataset as input and convert the data into object known as Patient

Moreover, function KNearestNeighbour will also call function findkNearest

```
def findKNearest(test_example, train_exampleSet, k=3):
    kNearest, dist_KNearest = [], []
    for i in range(k):
        kNearest.append(train_exampleSet[i])
        dist_KNearest.append(test_example.distance(train_exampleSet[i]))
    maxDist = max(dist_KNearest)
    for e in train_exampleSet[k:]:
        dist = test_example.distance(e)
        if dist < maxDist:
            maxIndex = dist_KNearest.index(maxDist)
            kNearest[maxIndex] = e
            dist_KNearest[maxIndex] = dist
            maxDist = max(dist_KNearest)
    return kNearest, dist_KNearest
```

Function findkNearest will take the input as the training and test data of the output of the function buildKidneyExample, and it will calculate the Euclidean distance and based in the calculation of Euclidean distance, this function will give the K Nearest Neighbours and their distances as output.

3. FITTING THE KNN ALGORITHM INTO THE DATASET

The implementation of KNN algorithm is done by the idea of abstraction i.e. importing the function KNeighborsClassifier from sklearn neighbors. The number of neighbours is given as input to this function. Afterwards, the model is fitted into the training data in order to train the algorithm and then this model is used in the test data

4. PERFORMANCE OF THE KNN ALGORITHM

The performance of the algorithm on the given training dataset is calculated on the following parameters:

1. True Positive
2. False Positive
3. True Negative
4. False Negative

These values are calculated by comparing the actual known data and the predicted data.

```
tp, tn, fp, fn = 0, 0, 0, 0
for j in range(len(predicted)):
    if predicted[j] == y_test[j] == 0:
        tp += 1
    if predicted[j] == y_test[j] == 1:
        tn += 1
    if predicted[j] == 0 and y_test[j] != predicted[j]:
        fp += 1
    if predicted[j] == 1 and y_test[j] != predicted[j]:
        fn += 1
print("\n" + "Count of true positive=", tp, " Count of true negative=", tn)
print("Count of false positive=", fp, " Count of false negative=", fn, "\n")
```

After obtaining the mentioned parameters, Accuracy, Positive Predicted Value, Negative Predicted Value Sensitivity, and Specificity of the algorithm is calculated by using the function getperformance.

```
def getPerformance(TruePositive, FalsePositive, TrueNegative, FalseNegative):
    Accuracy = accuracy(TruePositive, FalsePositive, TrueNegative, FalseNegative)
    Sensitivity = sensitivity(TruePositive, FalseNegative)
    Specificity = specificity(TrueNegative, FalsePositive)
    ppv = PositivePredictedValue(TruePositive, FalsePositive)
    npv = NegativePredictedValue(TrueNegative, FalseNegative)
    print(' Accuracy =', round(Accuracy, 3))
    print(' Sensitivity =', round(Sensitivity, 3))
    print(' Specificity =', round(Specificity, 3))
    print(' Pos. Pred. Val. =', round(ppv, 3))
    print(' Neg. Pred. Val. =', round(npv, 3), "\n")
    return (Accuracy, Sensitivity, Specificity, ppv, npv)
```

5. OUTPUT

```
Accuracy = 0.877
Sensitivity = 0.939
Specificity = 0.812
Pos. Pred. Val. = 0.838
Neg. Pred. Val. = 0.929
```

```
Finished processing 326 patients

[<__main__.Patient object at 0x00000169ACFCF508>, <__main__.Patient object at 0x00000169ACFCA308>, <__main__.Patient object at 0x00000169ACFC6248>]
[2.3430749027719964, 2.118962010041709, 1.2206555615733703]
```

DECISION TREE ALGORITHM

1. USING THE DECISION TREE ALGORITHM TO CHECK FOR KIDNEY DISEASE

A decision tree or a classification tree is a tree in which each internal (non-leaf) node is labeled with an input feature. The arcs coming from a node labeled with a feature are labeled with each of the possible values of the feature. Each leaf of the tree is labeled with a class or a probability distribution over the classes. [8]

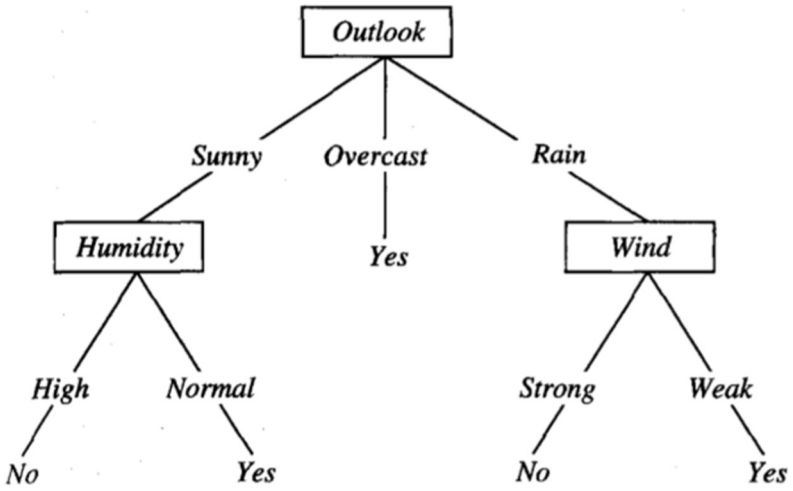


Fig 5. A decision tree for the concept PlayTennis

In the Fig 6. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, Yes or No), This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis [9].

The types of decision tree algorithms include:

- ID3 (Iterative Dichotomiser): To construct a decision tree, ID3 uses a top-down, greedy search through the given sets, where each attribute at every tree node is tested to select the attribute that is best for classification of a given set [10].
- C4.5: It generates decision trees which can be used for classification and therefore C4.5 is often referred to as statistical classifier [11]. It deals with both continuous and discrete attributes and also with the missing values and pruning trees after construction.
- CART (Classification and regression Trees): The tree is built from the root node by splitting using binary questions asked recursively at each node [13].

METHODOLOGY

1. DATA PREPROCESSING:

Given that some of the values in the data set are non-numerical, we convert them into numerical values using linear model preprocessing from sklearn library and use it to transform all of the data.

```
def process_split(dt, ts):  # function for processing the values and splitting into training and testing sets
    dtc = list(dt.columns.values)
    label = preprocessing.LabelEncoder()
    for i in dtc:
        dt[str(i)] = label.fit_transform(list(dt[str(i)]))
    xc = dt.values[:, 0:6]
    yc = dt.values[:, 6]
    xt, xts, yt, yts = sklearn.model_selection.train_test_split(xc, yc, test_size=ts)
    return xt, xts, yt, yts
```

Description:

- Here we iterate across the column titles and transform all of the corresponding values in each column.
- We then reassemble the data into the attributes and the output.
- We further split this into a training and a test data set for a chosen test size

2. DECISIONTREECLASSFIER FUNCTION CALL:

```
def DT_gini(xg, yg):  # function for criteria as gini
    clf_g = DecisionTreeClassifier(criterion="gini", random_state=100, max_depth=3, min_samples_leaf=5)
    clf_g.fit(xg, yg)
    return clf_g


def DT_entr(xe, ye):  # function for criteria as entropy
    clf_e = DecisionTreeClassifier(criterion="entropy", random_state=100, max_depth=3, min_samples_leaf=5)
    clf_e.fit(xe, ye)
    return clf_e
```

Description [12]:

- criterion: string, optional (default=”gini”)
  The function to measure the quality of a split.
- max_depth: int or None, optional (default=None)
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_samples_leaf : int, float, optional (default=1)
  The minimum number of samples required to be at a leaf node.
- random_state : int, RandomState instance or None, optional (default=None)
  If int, random_state is the seed used by the random number generator

3. MAIN FUNCTION CALL:

```
def realtime():   # Function for executing decision tree in real time
    kdata = pd.read_csv("Cleankidney.txt")
    datahead = list(kdata.columns.values)

    nc = input("Enter your choice of criteria: 0 for entropy, 1 for gini")
    if not (nc == "1") and not (nc == "0"):
        print("Invalid entry")
        exit()
    best = 0
    for t in range(5):  # to feed the pickle files over 30 iterations
        x_train, x_test, y_train, y_test = process_split(kdata, 0.2)
        y_pred, dclass = decisions(nc, x_train, y_train, x_test)
        acc = metrics.accuracy_score(y_test, y_pred)
        while acc > best:
            best = acc
            if nc == "0":
                with open("kidneyentropy.pickle", "wb") as f:
                    pickle.dump(dclass, f)
            else:
                with open("kidneygini.pickle", "wb") as f:
                    pickle.dump(dclass, f)

    names = ["ckd", "notckd"]

    acc = metrics.accuracy_score(y_test, y_pred)  # function for determining accuracy
    print(acc)

    tp, tn, fp, fn = metricfunc(y_test, y_pred)

    print("count of true positive=", tp, "count of true negative=", tn)
    print("count of false positive=", fp, "count of false negative=", fn)

    for x in range(len(y_pred)):
        print("predicted:", names[y_pred[x]], "Data:", x_test[x], "actual:", names[y_test[x]])
```

Description:

- We create the model on either gini or entropy as criterion as per the user choice and then fit the data to the model, after splitting into test and training data sets.
- We iterate 5 times and feed into the pickle model of either kidneyentropy.pickle or kidneygini.pickle. This allows us to use the saved model for the next set of values.
- We print out the metric values which will be discussed in the next section
- Finally, we print out the predicted values against the actual values for comparison with the accuracy

4. PERFORMANCE OF THE DECISION TREE ALGORITHM:

The performance of the algorithm on the given training dataset is calculated on the following parameters:

- True Positive: The training value is positive and the predicted value is positive.
- False Positive: The training value is negative and the predicted value is positive.
- True Negative: The training value is negative and the predicted vale is negative.
- False Negative: The training value is positive and the predicted value is negative

These values are calculated by comparing the actual known data and the predicted data.

```
def metricfunc(tested, predic):
    trpo = 0
    trne = 0
    flpo = 0
    flne = 0
    for i in range(len(predic)):
        if (predic[i] == 1) and (tested[i] == 1):
            trpo = trpo + 1

        elif (predic[i] == 1) and (tested[i] == 0):
            flpo = flpo + 1

        elif (predic[i] == 0) and (tested[i] == 0):
            trne = trne + 1

        else:
            flne = flne + 1
    return trpo, trne, flpo, flne
```

Description

- The training values are compared against the predicted values of the output.
- We save the final count of the four values.

With the obtained parameters we can evaluate the following parameters: Accuracy, Positive Predicted value, Negative Predicted Value Sensitivity and the Specificity of the algorithm.

```
def getPerformance(TruePositive, FalsePositive, TrueNegative, FalseNegative):
    Accuracy = accuracy(TruePositive, FalsePositive, TrueNegative, FalseNegative)
    Sensitivity = sensitivity(TruePositive, FalseNegative)
    Specificity = specificity(TrueNegative, FalsePositive)
    ppv = PositivePredictedValue(TruePositive, FalsePositive)
    npv = NegativePredictedValue(TrueNegative, FalseNegative)
    print(' Accuracy =', round(Accuracy, 3))
    print(' Sensitivity =', round(Sensitivity, 3))
    print(' Specificity =', round(Specificity, 3))
    print(' Pos. Pred. Val. =', round(ppv, 3))
    print(' Neg. Pred. Val. =', round(npv, 3), "\n")
    return (Accuracy, Sensitivity, Specificity, ppv, npv)
```

5. OUTPUT:

- Selected Criterion: Gini

```
Enter your choice of criteria: 0 for entropy, 1 for gini1
count of true positive= 30 count of true negative= 34
count of false positive= 2 count of false negative= 0
predicted: ckd Data: [ 7  3 37 14  0  0] actual: ckd
predicted: notckd Data: [24  0 35  8  0  0] actual: notckd
predicted: ckd Data: [36  0 27 16  1  0] actual: ckd
predicted: notckd Data: [11  0 10  3  0  0] actual: notckd
predicted: ckd Data: [38  1 23  8  1  1] actual: ckd
predicted: notckd Data: [28  0  6  1  0  0] actual: notckd
predicted: notckd Data: [23  0  6  4  0  0] actual: ckd
predicted: ckd Data: [56  2 56 28  0  0] actual: ckd
predicted: ckd Data: [47  0 34 12  1  1] actual: ckd
predicted: ckd Data: [ 5  3 14  5  0  0] actual: ckd
```

```
Accuracy = 0.939
Sensitivity = 0.892
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.879
```

- Selected Criterion: Entropy

```
Enter your choice of criteria: 0 for entropy, 1 for gini0
count of true positive= 29 count of true negative= 37
count of false positive= 0 count of false negative= 0
predicted: notckd Data: [21  0 18  4  0  0] actual: notckd
predicted: ckd Data: [40  2 41 33  1  1] actual: ckd
predicted: notckd Data: [65  0 36  3  0  0] actual: notckd
predicted: ckd Data: [69  2 51 29  1  1] actual: ckd
predicted: ckd Data: [57  2 41 12  1  1] actual: ckd
predicted: notckd Data: [59  0 29  8  0  0] actual: notckd
predicted: ckd Data: [44  3 57 44  1  1] actual: ckd
predicted: notckd Data: [25  0 18  8  0  0] actual: notckd
predicted: ckd Data: [54  3 47 24  1  1] actual: ckd
predicted: ckd Data: [43  1 10  6  0  1] actual: ckd
```

```
Accuracy = 0.97
Sensitivity = 0.946
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.935
```

PART 2: STUDY ON THE GROUND UP DESIGN OF THE CART (CLASSIFICATION AND REGRESSION TREES) ALGORITHM

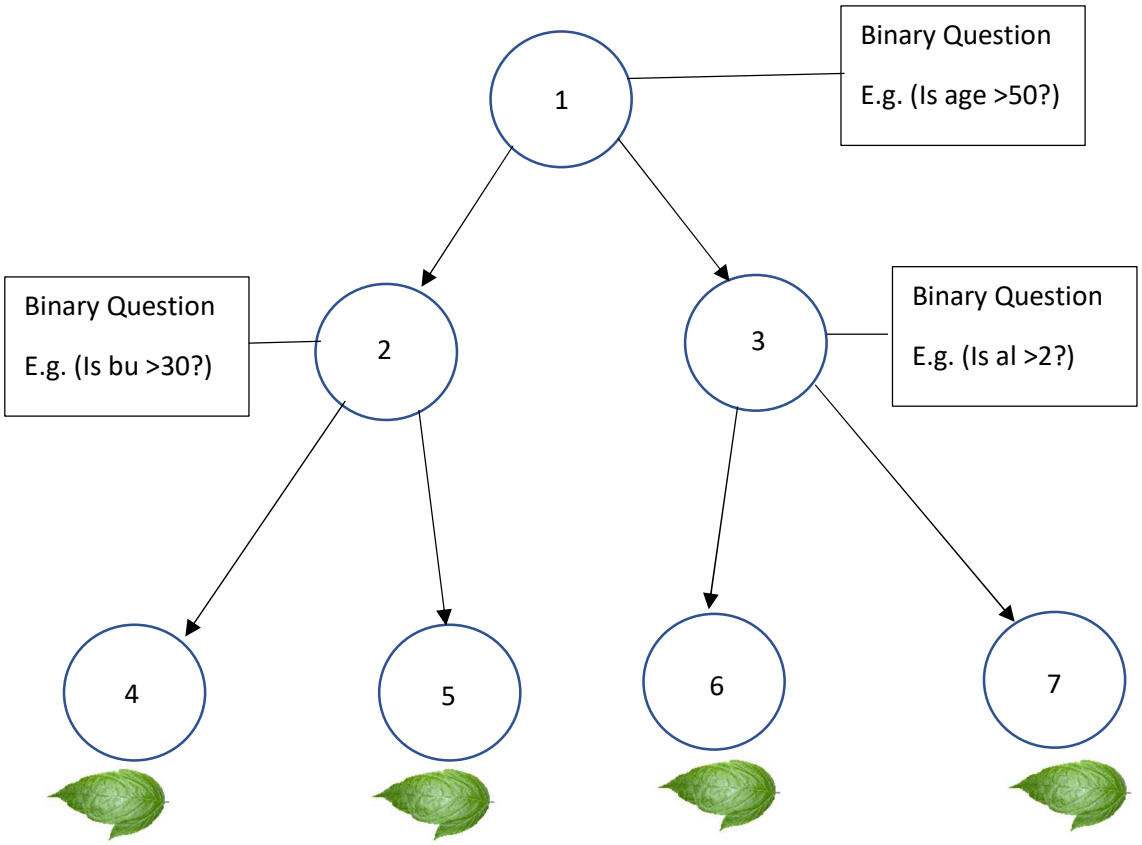


Fig 6. CART Tree

ALGORITHM [13]:

1. The best attribute (N) to split on, is evaluated using a test such as the Gini index (quantifies the measure of uncertainty at a single node) and information gain (quantifies measure of how much a question helps unmix the label).

2. The root of the tree would be the aforementioned best attribute (N).

3. For each value of N, we ask a yes or no question upon which it will be split into two leaf nodes. The subsets then become the input to child nodes we add to the tree.

4. The process is recursively continued by lining up all the values and using different split points using a cost function (Gini index) until we have a pure distribution of data.

METHODOLOGY

1. ASKING A QUESTION TO THE NODE:

```
class testingcondition:
    def __init__(self, col, colv):
        self.col = col
        self.colv = colv

    def match(self, example):
        val = example[self.col]
        if val > self.colv:
            return True
        else:
            return False

    def __str__(self):
        return 'Is ' + dtc[self.col] + ' > ' + str(self.colv)
```

Description:

- This class will deal with two variables, the column (col) and the values within the column (colv).
- When we pass a value into the class as example, the match function will check if the condition holds true for this value.
- In the following example, the testing condition is whether the first column, that is age, is greater than 50.
- The final return value would be false as tdata[0] = [38 1 23 8 1 1 0]

Output:

```
# q = testingcondition(0, 50)
# >>> Is age > 50
# example = tdata[0]
# r = q.match(example)
# print(r)
```

2. TO SPLIT THE DATA BASED ON THIS QUESTION:

```
def split(rw, tq):
    t_row, f_row = [], []
    for r in rw:
        if tq.match(r):
            t_row.append(r)
        else:
            f_row.append(r)
    return t_row, f_row
```

Description:

- We create two empty lists to feed the true values and the false values as determined by our condition.
- Here the testing condition is age>50.

Output:

```
# true_rows, false_rows = split(tdata, testingcondition(0, 50))
# true_rows = [array([52,  2, 39, 14,  0,  1,  0], dtype=int64),
#              array([58,  0, 40, 70,  0,  0,  0], dtype=int64)....
# false_rows = [array([38,  1, 23,  8,  1,  1,  0], dtype=int64),
#              array([4, 4, 5, 4, 0, 0, 0], dtype=int64)....
```

3. FINDING THE BEST QUESTION TO ASK A NODE:

For this purpose, we use two values, the Gini index gives us the measure of uncertainty at a node (the amount of mixing of labels), while the information gain gives us the measure of how much the uncertainty decreases between parent and child nodes.

Algorithm of the Gini Index:

1 .The Gini impurity can be computed by summing the probability of an item with label being chosen times the probability of a mistake in categorizing that item

2. Count the number of instances of each type of example in the dataset

3. Initially assign impurity as 1

3. Calculate probability of class in the given branch

4. Sum the squared class probabilities. Subtract the sum from 1

```
def gini(data):
    impurity = 1
    counts = cls_cnt(data)
    for t in counts:
        prob_val = counts[t] / float(len(data))
        impurity = impurity - prob_val**2
    return impurity
```

Description:

- Probability of a value is given by its count in the column (accessed within the dictionary) divided by the length of the data set.
- This value is subtracted from the impurity which was originally 1.

Output:

```
# b = gini(tdata)
# >>> 0.48628100417780123
```

Algorithm of information gain:

1. Measure the uncertainty of the starting set

2. We ask a binary equation and the measure the uncertainty of each of the two child nodes.

3. Here we will take a weighted average of the uncertainty as we seek to have a lower uncertainty over the whole tree instead of the converse.

4. We will subtract this uncertainty from that of the starting uncertainty and this gives us the information gain.

5. The question which gives us the most gain, would be the best question to ask at a node.

```
def info_gain(tbran, fbran, uct):

    p = float(len(tbran)) / (len(tbran) + len(fbran))
    ig = uct - p * gini(tbran) - (1 - p) * gini(fbran)
    return ig
```

Description:

- The info gain is given by the gini index of the starting node - the weighted gini index of the true branch - the weighted gini index of the false branch.

Output:

```
# a = gini(tdata)
# true_rows, false_rows = split(tdata, testingcondition(0, 50))
# b = info_gain(true_rows, false_rows, a)
# print(b)
# >>> 0.0341668057783549
```

4. ASKING THE QUESTION USING ALL THE UNIQUE VALUES IN EACH COLUMN

```
def find_best_split(rows):

    b_gain = 0
    b_ques = None
    crt_uncertainty = gini(rows)
    n_feat = len(rows[0]) - 1

    for col in range(n_feat):
        a = []
        for r in rows:
            a.append(r[col].item())
            values = set(a)
```

Output for values shown at this point

```
    for val in values:


          tques = testingcondition(col, val)
          trow, frow = split(rows, tques)

          if len(trow) == 0 or len(frow) == 0:
              continue

          gain = info_gain(trow, frow, crt_uncertainty)

          if gain >= b_gain:
              b_gain, b_ques = gain, tques

 return b_gain, b_ques
```

Description:

- Here we keep track of the best information gain and the feature / value that produced it.
- We iterate across each of the unique values and assign each as the condition to find best gini index and information gain.

Output:

```
# Innermost for loop output :
# {0, 1, 2, 3,......69, 70, 71}
# {0, 1, 2, 3, 4, 5}
# {0, 1, 2, 3.......98, 99, 100}
# {0, 1, 2, 3.......69, 70, 71}
# {0, 1}
# {0, 1}
```

5. INPUT THE DATA AND CALL THE RESPECTIVE FUNCTIONS

```
kdata = pd.read_csv("Cleankidney.txt")
dtc = list(kdata.columns.values)
label = preprocessing.LabelEncoder()
for i in dtc:
    kdata[str(i)] = label.fit_transform(list(kdata[str(i)]))
tdata = kdata.values[:, 0:7]

best_g, best_q = find_best_split(tdata)
print(best_g)
print(best_q)
```

Description:

- We input the data “Cleankidney.txt” file.
- We transform all the values iterating across the headings of each columns and assemble the data into tdata.
- We then call and display the best question and the best gain to ask the starting node.

OUTPUT:

```
# FINAL OUTPUT
# Is sc > 8
# 0.2858491571750251
```

STEP 6: We recursively call these functions at every node to create the tree.

STEP 7: We print the tree out and then prune it as needed to improve the accuracy of the predictions.

RELATED WORK

GUI INTRODUCTION

Graphical user interface (GUI) is a user-friendly form that allows users to easily interact with the program through the graphical labels or buttons etc. in order to achieve the goal without looking into the "black box" of the program. A GUI uses windows, icons, and menus to carry out commands, such as opening, deleting, and moving files. Since GUI is so convenient and modern, it is a dominant way in which the programs present.

GUI can be created with Python and used for Python programs to enable users to run and interact with codes. In order to create a GUI for our project, there are several external libraries or modules, and the module called PyQt5 was adopted. PyQt5 is integrated with the Qt designer, making it easier to build and beautify the interfaces. Besides, this module supports multiple OS, including Microsoft Windows and MacOS.

GUI OPERATIONS

To install all the elements of PyQt5, we can use pip commands:

- "pip install pyqt5" to install the main components
- "pip install pyqt5-tools" to install the Qt designer
- "pip install pyqt5-stubs" to install the stubs and add-ons

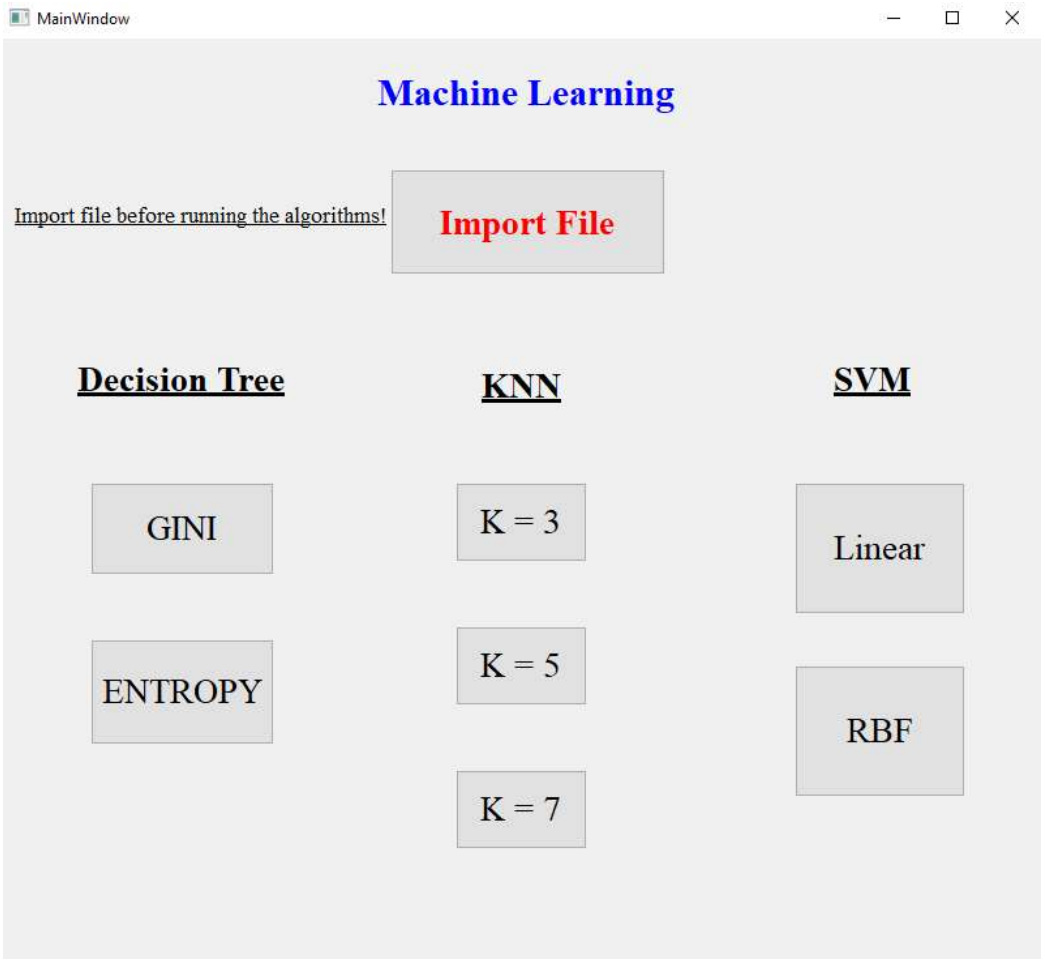
Once we have all of these installed, we can start writing the codes to create a GUI. For a basic interface, we only need to import QtWidgets from PyQt5, and import sys.

The interface for our project mainly consists buttons and label. It has a button for importing the file of input, and another six buttons for running the specific algorithms, as well as the labels are for information display. When the import button is clicked, the data file will be imported, and it is ready for executing the machine learning part. Also, a popup message window will come out saying the file has been successfully imported.

The main built-in objects that need to be used are QMainWindow, QLabel and QPushButton to create a main interface, labels and buttons respectively. There are considerable amounts of built-in functions linked to these built-in objects in PyQt5 module. We can use these them to alter and customize the style, size and content etc. by typing dot and the attributes after an object. For example, we can type "self.label1.setGeometry(…)" to set the size and location of a label. Because generally we use OOP for GUI construction, "self" is used to identify this is a private object.

If we want to execute a function or a class when the button is clicked, we can type "self.button1.clicked.connect(…)" to link this button to the specific function or class. Besides, for creating the popup message window mentioned above, a function called "msgbox1" was created inside the class with the built-in object QMessageBox created, and then we can construct and customize this popup message window by using the built-in functions. Then we can simply call up this message window by typing "self.button1.clicked.connect(self.msgbox1)".

OUTPUT



SEPARATE IMPORT FUNCTION

In order to enable users to change the input data file in a simpler way, a separate import function along with a button on GUI is created. When users want to change the input data file, they only need to click the import button, and type the file name in the console, and the program will know which file will be used for running the machine learning algorithms.

KEY INDEX AND METRIC SYSTEM

In order to compare the performance of different algorithms, we want to introduce the following indices:

- True Positive (TP): The predicted value and actual value match, and both are “True”.
- False Positive (FP): The predicted value and actual value do not match; predicts “True”, while actual is “False”.
- False Negative (FN): The predicted value and actual value do not match; predicts “False”, while actual is “True”.
- True Negative (TN): The predicted value and actual value match, and both are “False”.

In our case, the person who has chronic kidney disease (ckd) means “True”, while who is healthy means “False”.

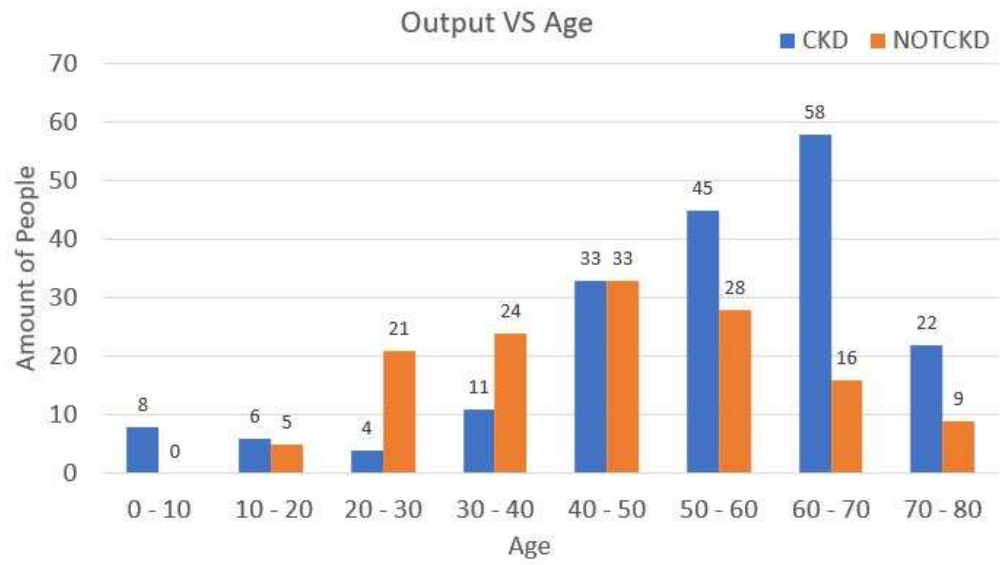
Based on these indices, we can use the following metrics:

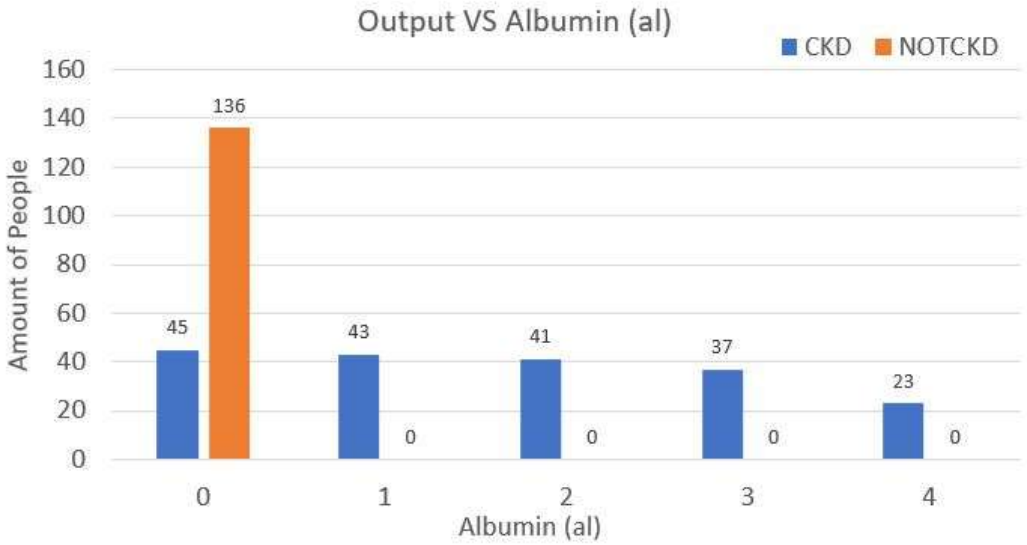
- Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$. If the predicted value matches the actual value, it is considered as accurate.
- Sensitivity = $\frac{TP}{TP+FN}$. This is to determine the proportion of true positive in total patients. (i.e. the ratio of patients that are accurately predicted among the total patients.)
- Specificity = $\frac{TN}{TN+FP}$. This is to determine the proportion of true negative in total healthy people. (i.e. the ratio of healthy people that are accurately predicted among the total healthy people.)
- Positive Predictive Value (PPV) = $\frac{TP}{TP+FP}$. This is the accuracy among the total positive predicted cases. (i.e. the ratio of patients that are accurately predicted among the total positive predicted cases.)
- Negative Predictive Value (NPV) = $\frac{TN}{TN+FN}$. This is the accuracy among the total negative predicted cases. (i.e. the ratio of healthy people that are accurately predicted among the total negative predicted cases.)

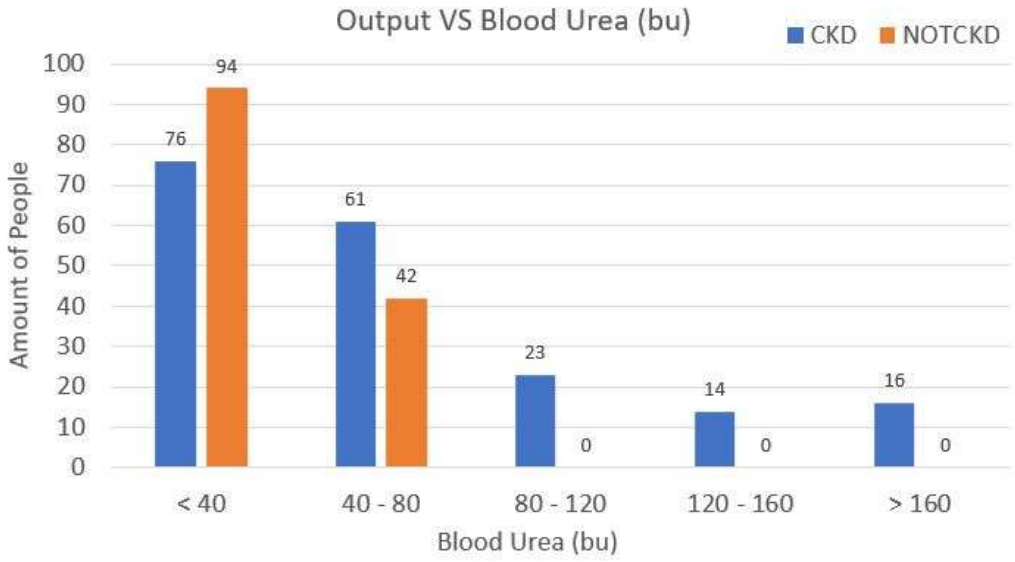
RESULTS

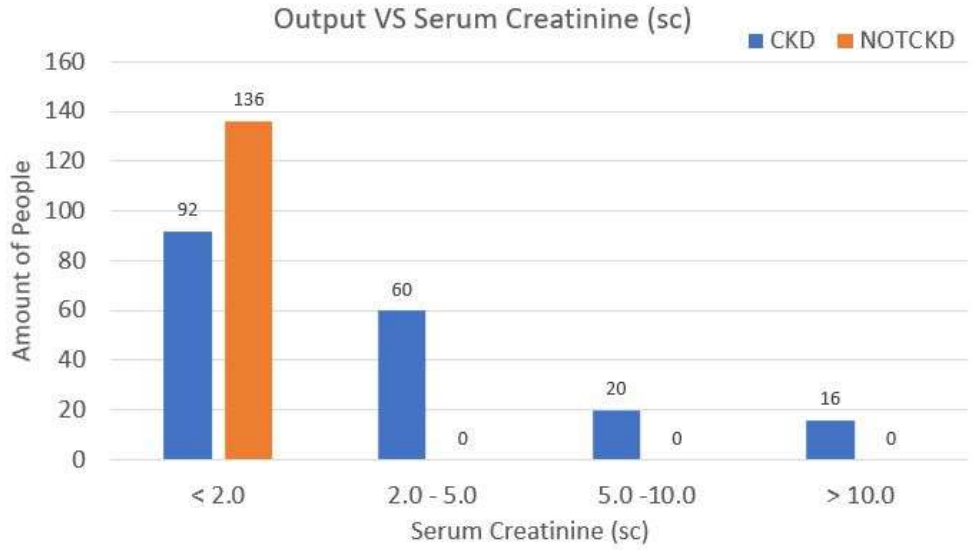
1. WITH REGARD TO THE DATASET

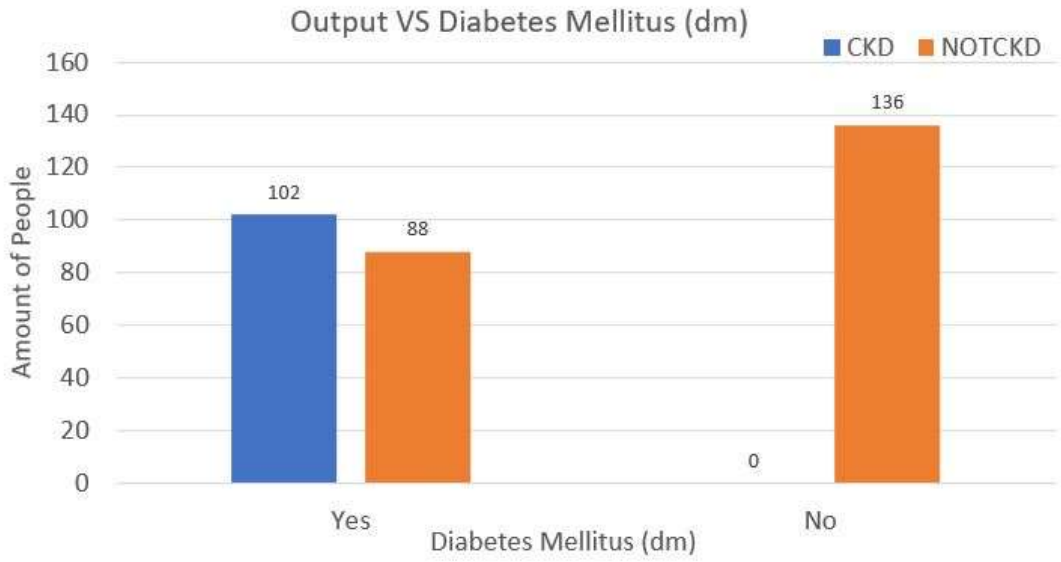
The following are the plotted graphs of various attributes against the output:

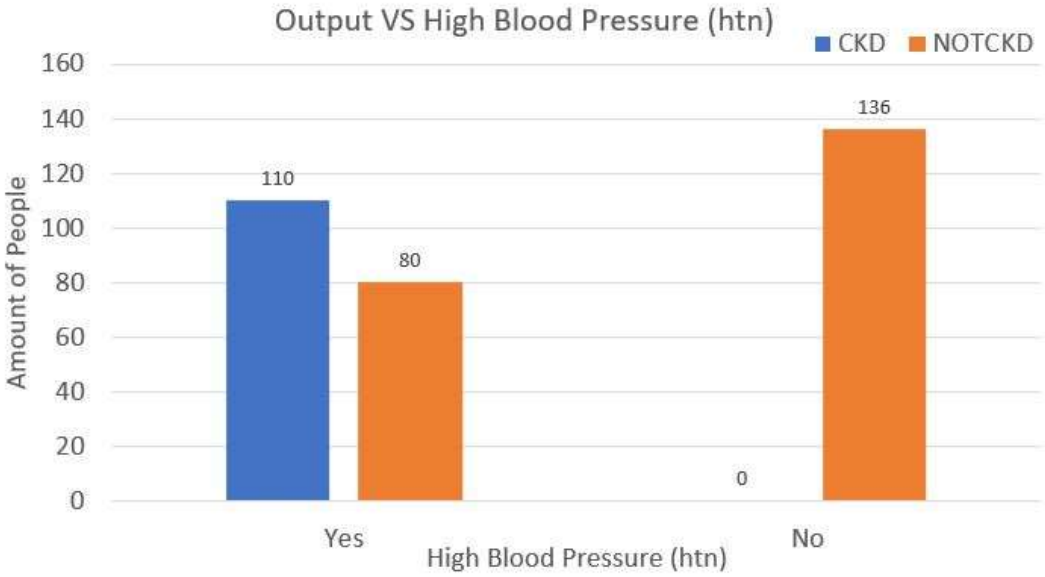





Output VS Blood Urea (bu)
CKD
NOTCKD
Amount of People
100
90
80
70
60
50
40
30
20
10
0
94
76
61
42
23
0
14
0
16
0
< 40
40 - 80
80 - 120
120 - 160
> 160
Blood Urea (bu)


Output VS Serum Creatinine (sc)
CKD
NOTCKD
Amount of People
160
140
120
100
80
60
40
20
0
136
92
60
0
20
0
16
0
< 2.0
2.0 - 5.0
5.0 -10.0
> 10.0
Serum Creatinine (sc)


Output VS Diabetes Mellitus (dm)
CKD
NOTCKD
160
140
120
100
80
60
40
20
0
Amount of People
102
88
0
136
Yes
No
Diabetes Mellitus (dm)


Output VS High Blood Pressure (htn)
CKD
NOTCKD
160
140
120
100
80
60
40
20
0
Amount of People
110
80
0
136
Yes
No
High Blood Pressure (htn)

WITH REGARD TO THE ALGORITHMS: METRICS

1. KNN

```
Accuracy = 0.877
Sensitivity = 0.939
Specificity = 0.812
Pos. Pred. Val. = 0.838
Neg. Pred. Val. = 0.929
```

```
Processing Time:  0.062816 seconds
```

K = 3

```
Accuracy = 0.8
Sensitivity = 0.85
Specificity = 0.72
Pos. Pred. Val. = 0.829
Neg. Pred. Val. = 0.75
```

```
Processing Time:  0.060862 seconds
```

K = 5

```
Accuracy = 0.769
Sensitivity = 0.69
Specificity = 0.913
Pos. Pred. Val. = 0.935
Neg. Pred. Val. = 0.618
```

```
Processing Time:  0.061833 seconds
```

K = 7

2. SVM

```
Accuracy = 0.909
Sensitivity = 0.973
Specificity = 0.828
Pos. Pred. Val. = 0.878
Neg. Pred. Val. = 0.96
```

```
Processing Time:  0.045886 seconds
```

KERNEL = RBF

```
Accuracy = 0.985
Sensitivity = 0.974
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.964
```

```
Processing Time:  0.103047 seconds
```

KERNEL = LINEAR

3. DECISION TREE

```
Accuracy = 0.97
Sensitivity = 0.946
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.935
```

```
Processing Time:  0.850548 seconds
```

CRITERION = GINI

```
Accuracy = 0.939
Sensitivity = 0.892
Specificity = 1.0
Pos. Pred. Val. = 1.0
Neg. Pred. Val. = 0.879
```

```
Processing Time:  0.014035 seconds
```

CRITERION = ENTROPY

CONCLUSION:

WITH REGARD TO THE DATA SET:

- The greatest incidence of kidney disease is over the age groups of 50 – 80 years of age.
- The greatest incidence of a healthy kidney is for an albumin value of 0.
- The greatest incidence of kidney disease is for a blood urea of 40 and lower.
- The greatest incidence of kidney disease if for a serum creatine of 2 and lower.
- The greatest incidence of kidney disease is with the presence of diabetes mellitus.
- The greatest incidence of kidney disease is with the presence of high blood pressure.

WITH REGARD TO THE ALGORITHMS:

- The linear kernel for SVM has the greatest accuracy followed by decision tree with criterions, Gini and entropy.
- The least accuracy is for KNN which peaks at K =3.
- In terms of processing time, the decision tree with a criterion of entropy is the fastest followed by SVM with a kernel of RBF, followed by KNN for which all the K values return similar runtimes.
- The slowest are decision tree with a criterion of Gini and SVM with a kernel of linear.

REFERENCES

[1] Malik, Usman. “Implementing SVM and Kernel SVM with Python's Scikit-Learn.” Stack Abuse. Stack Abuse, May 9, 2019. https://stackabuse.com/implementing-svm-and-kernel-svm-with-pythons-scikit-learn/.

[2] MLMath.io. “Math behind Support Vector Machine (SVM).” Medium. Medium, February 16, 2019. https://medium.com/@ankitnitjsr13/math-behind-support-vector-machine-svm.

[3] https://www.kidney.org/kidneydisease/global-facts-about-kidney-disease

[4] Devroye, L. & Wagner, T.J. (1982) "Nearest neighbor methods in discrimination, In Classification, Pattern Recognition and Reduction of Dimensionality", Handbook of Statistics, 2: 193–197. North-Holland, Amsterdam.

[5] Domeniconi, C., Peng, J. & Gunopulos, D. (2002) "Locally adaptive metric nearestneighbor classification", IEEE Transactions on Pattern Analysis and Machine Intelligence. 24(9): 1281–1285.

[6] Dudani, S.A. (1976) "The distance-weighted k-nearest neighbor rule", IEEE Transactions on System, Man, and Cybernetics, 6: 325-327.

[7] Eldestein, H.A. (1999) "Introduction to Data Mining and Knowledge Discovery", Two Crows Corporation, USA, ISBN: 1-892095- 02-5.

[8] Enas, G.G. & Choi, S.C. (1986) "Choice the smoothing parameter and efficiency of KNearest Neighbor classification", Comp & Maths with Apps, 12(2): 235-244.

[9] Fix, E. & Hodges, J.L. (1951) "Nonparametric Discrimination: Consistency Properties Randolph Field, Texas, Project 21-49-004, Report No. 4

[8] Poole, David & Mackworth, Alan “Learning Decision Trees”, Artificial Intelligence, LCI, UBC, Vancouver, Canada
https://artint.info/html/ArtInt_177.html

[9] Mitchell, M., Tom. (1997) “Decision Tree Learning”, Machine Learning, p. 53.

[10] T.Miranda Lakshmi, A.Martin, R.Mumtaj Begum and Dr.V.Prasanna Venkatesnan,“An Analysis on Performance of decision Tree Algorithms using Student’s Qualitative Data”, I.J.Modern Education and Computer Science, June 2013.

[11] Ian H. Witten; Eibe Frank; Mark A. Hall (2011). "Data Mining: Practical machine learning tools and techniques, 3rd Edition". Morgan Kaufmann, San Francisco. p. 191.

[12]” sklearn.tree.DecisionTreeClassifier” Scikit Learn.
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[13] Gordon, Josh “Let’s Write a Decision Tree Classifier from Scratch”, Machine Learning Recipes #8, https://www.youtube.com/watch?v=LDRbO9a6XPU&list=PL9w3LeXmj_eZE6XR01n6VSQvMm1TLbjgH&index=2&t=0s