Western Engineering

Using Machine Learning to Determine Kidney Disease

Group 4

- Xiangliang Yu
- Uriev Ellapen
- Tanveer Alam
- Sathyanath Nandakumar

Predict if the patient has chronic kidney disease (ckd) based on the 6 selected attributes using 3 different algorithms.

- **Support Vector Machine (SVM)**
- **K-Nearest Neighbor (KNN)**
- **Decision Tree**

6 Attributes

- **Age**
- **Albumin (AL)**
- **Blood Urea (BU)**
- **Serum Creatinine (SC)**
- **High Blood Pressure (HTN)**
- **Diabetes Mellitus (DM)**

We use a function to transform the data into numerical vectors and take 80% of the data for training and 20% for testing.

Western Engineering

Key Index from the Output

Predicted Value (predicted by the test)		Actual Value (as confirmed by experiment): positives	Actual Value (as confirmed by experiment): negatives
	positives	**TP** True Positive	**FP** False Positive
	negatives	**FN** False Negative	**TN** True Negative

- **TP: Predicted = Positive, & Actual = Positive**
- **FP: Predicted = Positive, & Actual = Negative**
- **FN: Predicted = Negative, & Actual = Positive**
- **TN: Predicted = Negative, & Actual = Negative**

In our case, the person who has chronic kidney disease (ckd) means Positive.

We can later compare the algorithms by the following metrics system:

- **Accuracy =** $\frac{TP+TN}{TP+TN+FP+FN}$
- **Sensitivity =** $\frac{TP}{TP+FN}$
- **Specificity =** $\frac{TN}{TN+FP}$
- **Positive Predictive Value (PPV) =** $\frac{TP}{TP+FP}$
- **Negative Predictive Value (NPV) =** $\frac{TN}{TN+FN}$

		Actual Value (as confirmed by experiment)	
		positives	negatives
Predicted Value (predicted by the test)	positives	**TP** True Positive	**FP** False Positive
	negatives	**FN** False Negative	**TN** True Negative

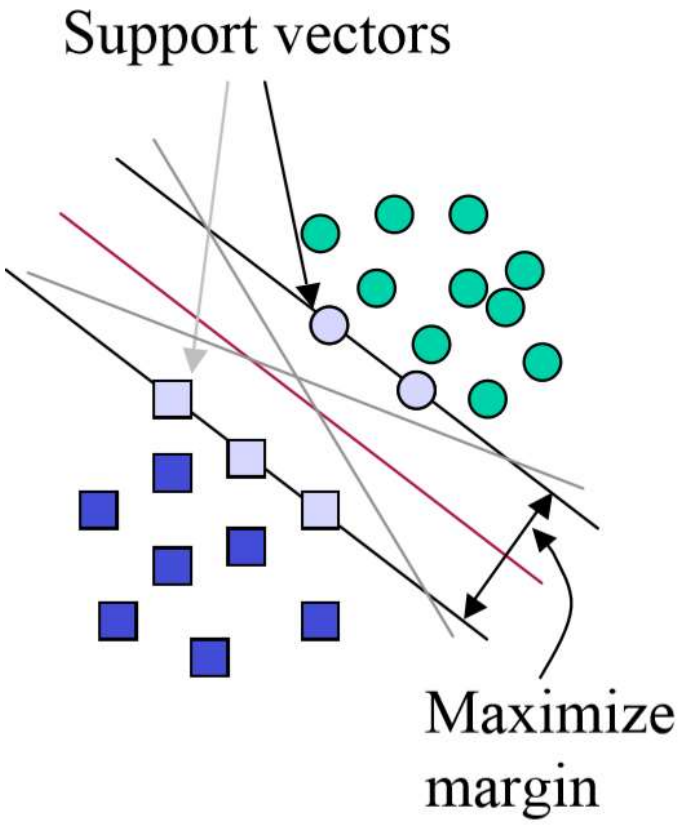
Support Vector Machine (SVM)

Machine learning algorithm that finds the optimal hyperplane to separate data using support vectors.

SVM algorithm is a constrained optimization problem that is solved using Lagrange multipliers.

SVM maximizes the margin around the separating hyperplane.

Kernel function is used to transform non-linear separable patterns into new space.

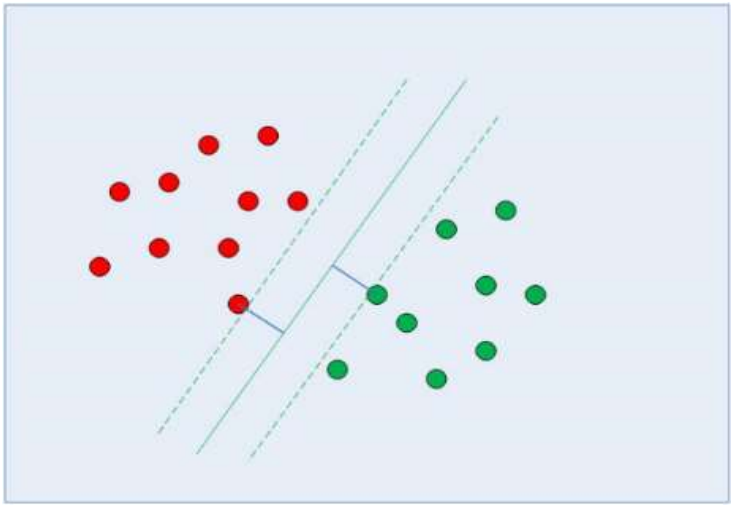


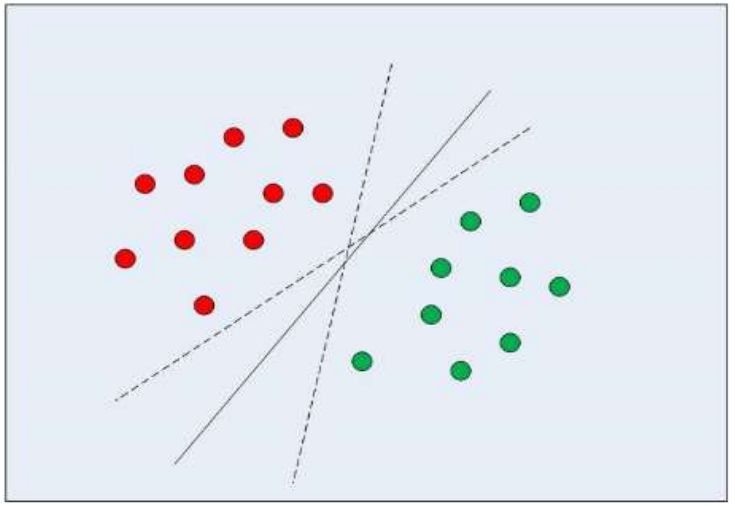
Basic Idea

- Support Vectors are the data points that lie closest to the hyperplane.
- They are the data points most difficult to classify and have direct bearing on the optimum location of the hyperplane
- Orientate hyperplane as far as possible from the closest support vectors of both classes

Linear Separability - 2 Dimensions

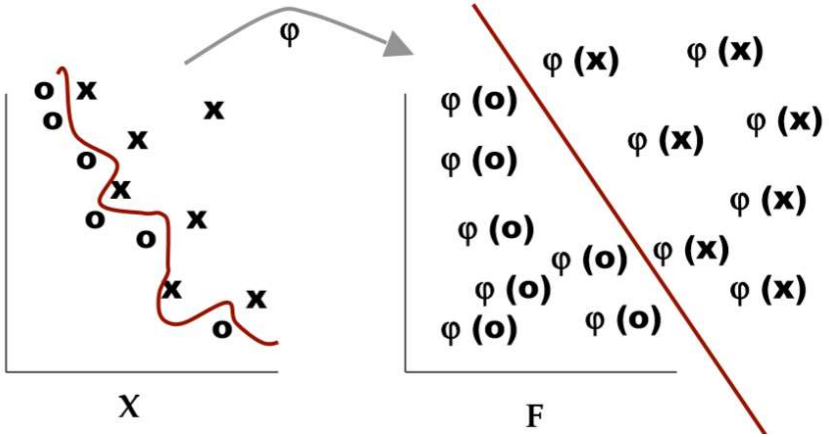
- Multiple hyperplanes are created with support vectors that are the two closest points to the line and equal in distance.
- SVM chooses hyperplane with largest margin between support vectors





Kernel For Non-Linear Separability

- Kernels map non-separable data into a higher dimension so that it may be divided using a hyperplane.



Transformation of data

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y}+1)^p$$

$$K(\mathbf{x},\mathbf{y}) = \exp\left\{-\|\mathbf{x}-\mathbf{y}\|^2 / 2\sigma^2\right\}$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(\kappa\mathbf{x}\cdot\mathbf{y}-\delta)$$

1st is polynomial (includes x•x as special case)
2nd is radial basis function (gaussians)
3rd is sigmoid (neural net activation function)

Non-Linear SVMs

KNN Approach

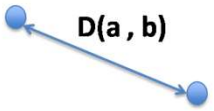
The simplest machine learning algorithm is the *k*-NN algorithm.

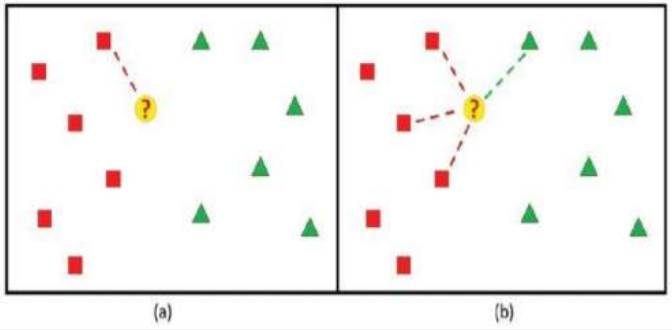
KNN stores all available cases and classifies new case based on distance measurement.

K is the number of neighbors considered for determining the new case.

```
def EuclideanDistance(p1, p2):
    dist = 0.0
    for j in range(len(p1)):
        dist += abs(p1[j] - p2[j]) ** 2
    return math.sqrt(dist)
```

$$D(a,b) = \sqrt{\sum_{i=1}^{n} (b_i - a_i)^2}$$



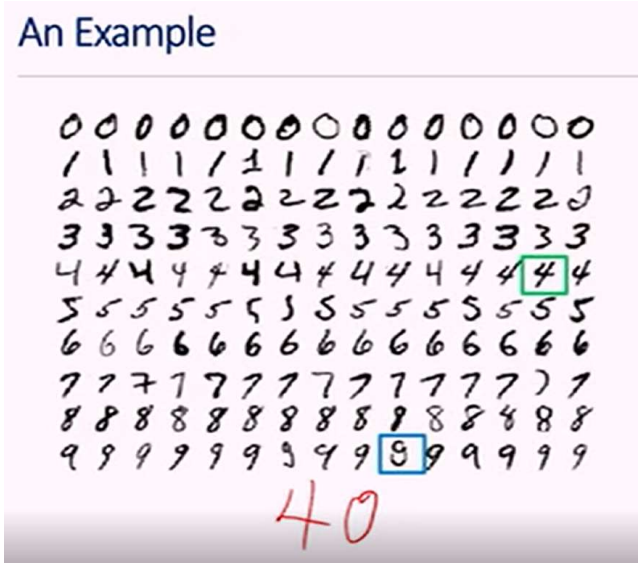


Basic Idea

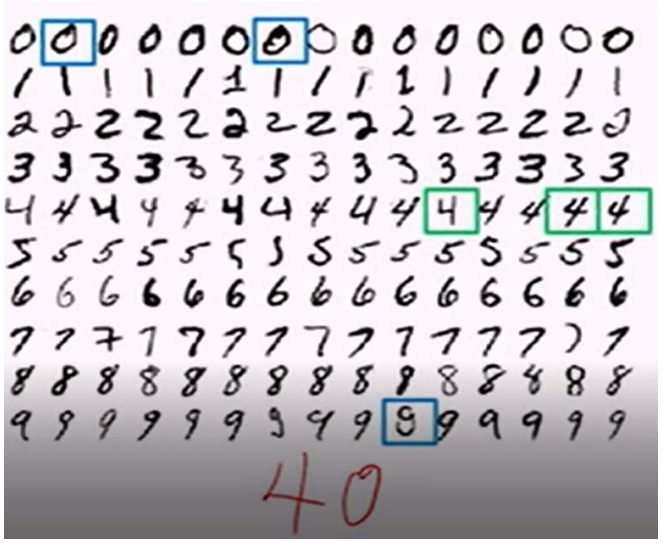
- *K*-NN classification rule is to assign to a test sample the majority category label of its *k* nearest training samples.
- The K Nearest Neighbors are computed by using Euclidean Distance Function .
- KNN assumes that all instances are points in n-dimensional space. The distance of the new/ test case is calculated from all the training examples, and the neighbours are defined in terms of nearest distance
- In practice, *k* is usually chosen to be odd, so as to avoid ties

How to Choose K ?

- If K is too small, it is sensitive to noise

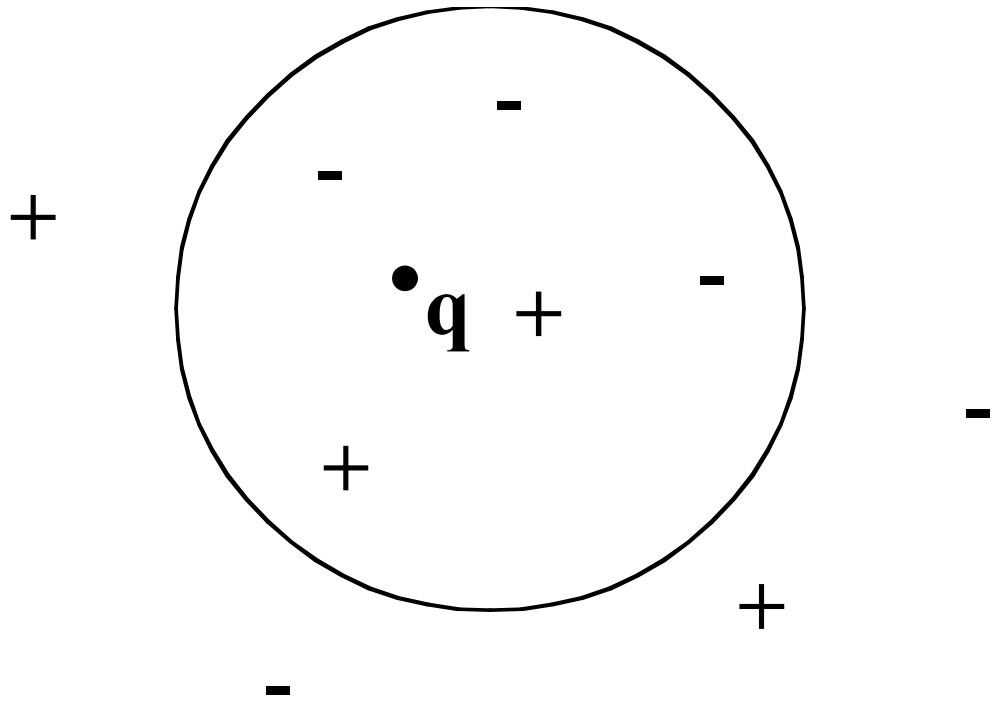


- Larger K works well



- Rule of thumb is K< sqrt(n), n is the number of examples

Simple Illustration



q is + under 1-NN, but – under 5-NN

KNN Algorithm for Chronic Kidney Disease

```
def KNearestClassifer3(k=3):
    start = time.time()
    data = file.import_files2()
    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"]))
    predict = "class"
    x = list(zip(age, al, bu, sc, htn, dm))
    y = list(cls)
    x_train, x_test = split(x)
    y_train, y_test = split(y)
    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(x_train, y_train)
    predicted = model.predict(x_test)
    names = ["ckd", "notcdk"]
    acc = metrics.accuracy_score(y_test, predicted)
    print("Accuracy:", acc, "\n")
    for i in range(len(predicted)):
        print("Predicted: ", names[predicted[i]], " Data: ", x_test[i], " Actual: ", names[y_test[i]])
    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")
    getPerformance(tp, fp, tn, fn)
    KNearestNeighbours(k=3)
    end = time.time()
    print("Processing Time: ", round(end - start, 6), "seconds","\n")
    return None
```

Outputs

```
Predicted:  ckd 0 Data:  (62, 0, 13, 5, 0, 0) Actual:  not cdk 1
Predicted:  not cdk 1 Data:  (51, 0, 25, 6, 0, 0) Actual:  not cdk 1
Predicted:  not cdk 1 Data:  (48, 0, 3, 7, 0, 0) Actual:  not cdk 1
Predicted:  ckd 0 Data:  (54, 0, 14, 3, 0, 0) Actual:  not cdk 1
Predicted:  not cdk 1 Data:  (68, 0, 32, 3, 0, 0) Actual:  not cdk 1

TruePositive 26 TrueNegative 23 FalsePositive 4 FalseNegative 12
```

```
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]
```

```
TruePositive 26 TrueNegative 23 FalsePositive 4 FalseNegative 12

 Accuracy = 0.754
 Sensitivity = 0.684
 Specificity = 0.852
 Postive Predicted Value = 0.867

Finished processing 326 patients
```

Strengths of KNN

- Very simple and intuitive
- Good Classification of the number of samples are large

Weakness of KNN

- Takes more time to classify a new example i.e. need to calculate and compare distance with all the other

```
Processing Time:  0.077791 seconds
```

- Need larger number of samples for accuracy

Decision Tree Algorithm

Decision tree learning serves as one of the most accurate supervised machine learning algorithms. It shows the particular resilience to noise and has multiple variants to suit particular tasks.

The aim is to produce an inverted tree with the end leaf nodes having a solely pure set of values (unmixed).

In general we will use an index to measure the uncertainty at a node and an index to measure how much it has reduced between levels.

The final tree can be pruned to prevent overfitting.

Part 1 : Project Code (Function calls)

- Processing the input data

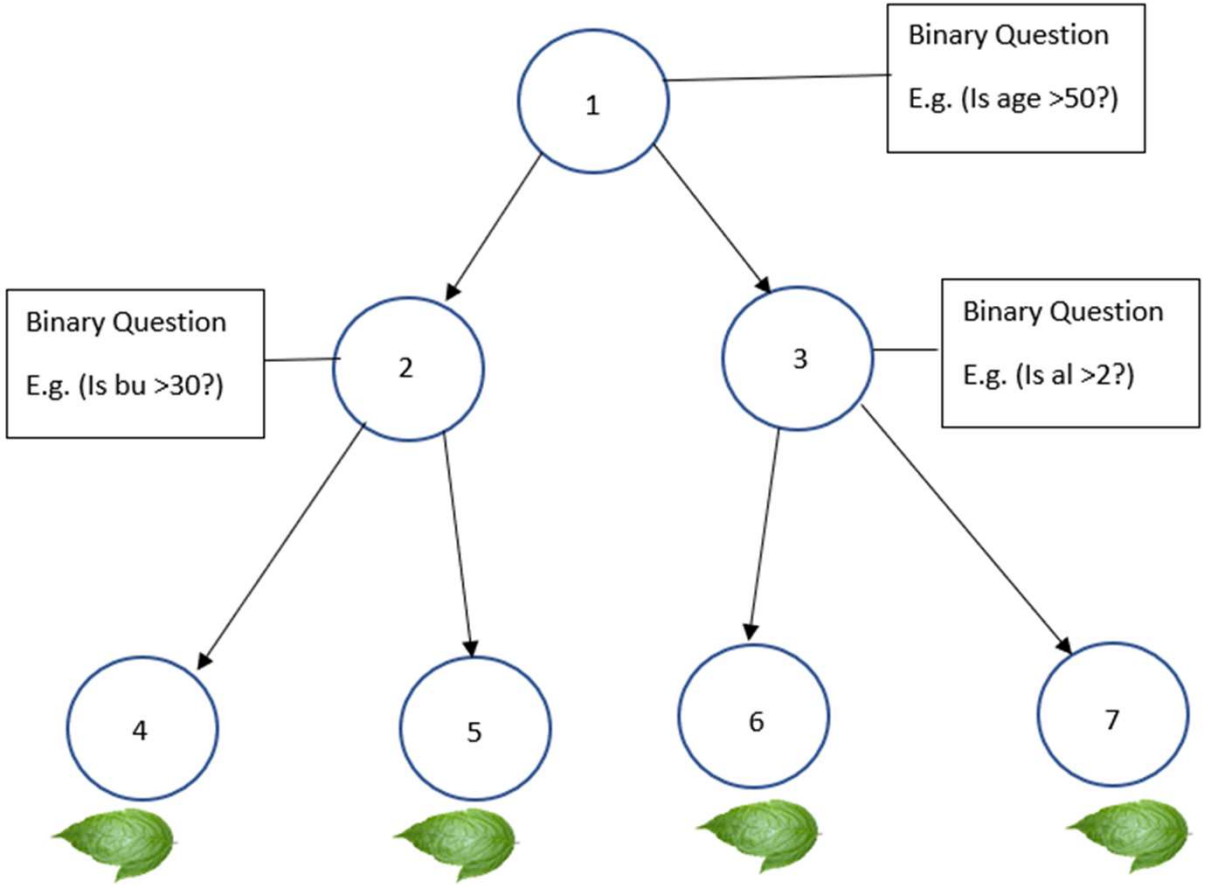
```
def process_split(dt, ts):  # function for processing the values and splitting into training and testing sets
    dtc = list(dt.columns.values)
    label = preprocessing.LabelEncoder()
    for i in dtc:
        dt[str(i)] = label.fit_transform(list(dt[str(i)]))
    xc = dt.values[:, 0:6]
    yc =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

- Gini and entropy 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
```

Part 2 : CART Algorithm (Binary Recursive Partitioning) Ground up Build



How to Ask a Question to a Node?

- Asking a question to a 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)
# q = testingcondition(0, 50)
# >>> Is age > 50
# example = tdata[0]
# here we access the first list within the list
# : [38  1 23  8  1  1  0]
# r = q.match(example)
# we check the whether the first value, age satisfies the condition
# print(r)   returns a false value
```

- Splitting data on its basis

```
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

# We create two empty lists to feed the true values
# and the false values as determined by our condiiton

# here the testing condition is age>50
# 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)....
```

How to Find the Best Question to Ask a Node?

- Gini Index

```
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

# probability of a value is given by
# its count in the column (accessed within the dictionary)
# divided by the len of the data set
# This value is subtracted from the
# impurity which was originally 1

# b = gini(tdata)
# >>> 0.48628100417780123
# (the gini value for the original data set)
```

```
def cls_cnt(items):
    cnt = dict()
    for c in items:
        out = c[-1]
        cnt[out] = cnt.get(out, 0) + 1
    return cnt
# creates a dictionary where the keys are the unique terms
# in the column and the values
# they relate to are the counts of that term
# get(key,value to return if key doesnt exist)
# method returns the value of the item with the specified key

# {0: 1}
# {0: 2}
# {0: 3}
# ...
# {0: 190, 1: 1}
# {0: 190, 1: 2}
# {0: 190, 1: 3}
# ...
# final result :  {0: 190, 1: 136}
```

Information Gain

```
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

# 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

# a = gini(tdata) gives the gini index of the starting dataset
# true_rows, false_rows = split(tdata, testingcondition(0, 50))
# we split the starting data set across the condition age>50
# b = info_gain(true_rows, false_rows, a)
# this is a measure of how much uncertainity has been reduced by that split
# print(b)
# >>> 0.0341668057783549
```

Finding the Best Split across All the Values

```
def find_best_split(rows):

    b_gain = 0  # keep track of the best information gain
    b_ques = None  # keep train of the feature / value that produced it
    crt_uncertainty = gini(rows)
    n_feat = len(rows[0]) - 1  # number of columns

    for col in range(n_feat):                # This inner for loop is to obtain the unique values
        a = []                               # in each column output:
        for r in rows:                       # {0, 1, 2, 3,......69, 70, 71}
            a.append(r[col].item())          # {0, 1, 2, 3, 4, 5}
            values = set(a)                  # {0, 1, 2, 3.......98, 99, 100}
                                             # {0, 1, 2, 3.......69, 70, 71}
                                             # {0, 1}
                                             # {0, 1}

       for val in values:  # we iterate along those values and
                           # assign each as the condition to find best gini index and info gain

            tques = testingcondition(col, val)
            trow, frow = split(rows, tques)   # try splitting the dataset

            if len(trow) == 0 or len(frow) == 0:  # Skip this split if it doesn't divide the dataset
                continue

            gain = info_gain(trow, frow, crt_uncertainty)

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

    return b_gain, b_ques
```

Main()

```
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]

# we transform the values into integers/ float using label encoder
# across all the column values and then feed them into tdata

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

# we print out the best gain and the best question to ask the first node
```

```
# FINAL OUTPUT
# Is sc > 8
# 0.285849157175025 1
```