



K-Nearest-Neighbor Decision Tree



kNearestNeighbor

- Lazy learner: defers model construction until test tuple arrives
- Based on learning by analogy
- Compares a given test tuple with all the training tuples
- Searches for the k training tuples that are **closest** to the unknown tuple
- ‘Closeness’ measure - a distance metric such as Euclidean distance between two tuples



kNearestNeighbor

- For classification:
 - 'k' closest points are selected
 - test tuple's class: the majority class appearing among k neighbors
- For regression:
 - 'k' closest points are selected
 - test tuple's prediction: avg. of labels associated with k neighbors



Drawbacks

- Computationally expensive for large training data
- Choice of k depends upon the training tuples
- Uses distance-based comparisons: need to convert binary, ordinal, and nominal data to numeric data
- Data normalization required to cancel the effects of large differences in attribute values



Algorithm

- m - the number of training data samples; p - an unknown point.
- Store the training samples in an array of tuple (x, y) .
- for $i=0$ to m :
 - Calculate Euclidean distance $d(arr[i], p)$.
- Obtain a set S of K smallest distances obtained and return the majority label among S .



Iris Dataset

- Contains 50 samples from each of three species of Iris
- They are **Iris setosa**, **Iris virginica**, and **Iris versicolor**
- Four features describe each class
- The features are sepal length, sepal width, petal length, and petal width

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa



Implementation

```
# distance between testInstance and each trainingSet
for i in range(len(trainingSet)):

    dist = manhattanDistance(testInstance, trainingSet.iloc[i], length)
    dist = euclideanDistance(testInstance, trainingSet.iloc[i], length)

    distances[i] = dist
```

```
sorted_d = sorted(distances.items(), key=lambda kv: kv[1])

# get k neighbors closest to the testInstance
neighbors = []

for i in range(k):
    neighbors.append(sorted_d[i][0])

# count of classes among neighbors
classVotes = {}

for x in range(len(neighbors)):

    class_label = trainingSet.iloc[neighbors[x]][-1]

    if class_label in classVotes:
        classVotes[class_label] += 1
    else:
        classVotes[class_label] = 1

sorted_v = sorted(classVotes.items(), key=lambda kv: kv[1], reverse=True)
```




Output

- With the value of $k = 1$ (no. of nearest neighbors); accuracy was about: 94.28571428571428%

```
train, test = load_iris2()  
KNearestNeighbor(train, test, 1)
```

```
For tuple 0; correct label is Iris-setosa; predicted lable is Iris-setosa
```

```
For tuple 1; correct label is Iris-setosa; predicted lable is Iris-setosa
```

```
For tuple 2; correct label is Iris-virginica; predicted lable is Iris-virginica
```

```
For tuple 3; correct label is Iris-setosa; predicted lable is Iris-setosa
```

```
Error ***** For tuple 4; correct label is Iris-virginica;  
predicted lable is Iris-versicolor
```

```
# varying the value of k

list_k = [1, 3, 5, 10, 20, 50, 100]
def test_for_k(train, test):

    accuracy = []
    for k in list_k:
        test_len = test.shape[0]

        correct = 0
        for i in range(test_len):

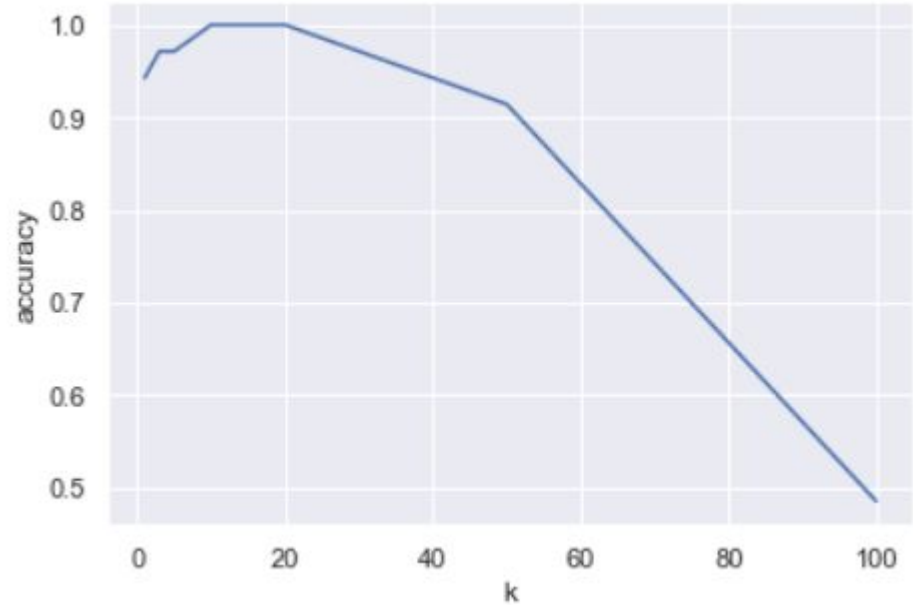
            result, neighbors = KNN(train, test.iloc[i][:-1], k)
            if result == test.iloc[i]['Name']:
                correct += 1

        accuracy.append(correct / test_len)

plt.xlabel('k')
plt.ylabel('accuracy')
plt.plot(list_k, accuracy)
```

Varying k

- Accuracy was found maximum when value of k was between 10 and 20





Decision tree

- Flowchart-like tree structure, with internal node making a decision, and leaf node holds a classification of a tuple
- Are easy to understand and interpret
- Can handle both categorical and continuous data
- Provides clear indication of prominent field for classification/prediction using attribute selection measures like information gain, gini index, etc.



Drawbacks

- Less appropriate for training with many classes and less examples
- Computationally expensive for large datasets
- No incremental training
- Unstable: small change in data leads to large change in the output of decision tree



Algorithm

- Find the best splitting criteria for the entire dataset
- Label root node with splitting criteria
- For each outcome k of splitting criteria
 - Find tuples satisfying k
 - If the size of tuples is empty, attach a leaf with majority class
 - Else attach the node returned by calling the algorithm with the remaining attribute list recursively



Implementation

Partition the dataset based on the question

i.e. Is SepalLength \geq 1.0 ?

Into two distinct partitions

```
def partitions(rows, question):  
    """Partition the dataset into 'true rows' or  
    'false rows' based on the matching with question  
    """  
  
    true_rows, false_rows = [], []  
    for row in rows:  
        if question.match(row):  
            true_rows.append(row)  
        else:  
            false_rows.append(row)  
    return true_rows, false_rows
```



Implementation

Calculate the Gini impurity for a given split of dataset

$$\text{i.e. } = 1 - \sum_{i=1}^J p_i^2$$

```
def gini(rows):  
    """Calculates the Gini Impurity for a list of rows"""  
  
    counts = class_counts(rows)  
    impurity = 1  
    for lbl in counts:  
        prob_lbl = counts[lbl] / float(len(rows))  
        impurity -= prob_lbl**2  
    return impurity
```

```
mixed_dataset = [['Iris-versicolor'],  
                  ['Iris-versicolor'],  
                  ['Iris-setosa'],  
                  ['Iris-versicolor'],  
                  ['Iris-versicolor']]  
  
gini(mixed_dataset)
```

0.31999999999999984



Implementation

The information gain from the split based on a certain value of the attribute

```
def info_gain(left, right, current_uncertainty):  
    """  
    Calculate information gain i.e.  
    uncertainty of starting node minus the weighted impurity  
    of two child nodes  
    """  
    p = float(len(left)) / (len(left) + len(right))  
    return current_uncertainty - p*gini(left) - (1-p)*gini(right)
```

```

best_gain = 0
best_question = None
current_uncertainty = gini(rows)
n_features = len(rows[0]) - 1

for col in range(n_features):

    values = set([row[col] for row in rows])

    for val in values:

        question = Question(col, val)
        true_rows, false_rows = partitions(rows, question)

        if len(true_rows) == 0 or len(false_rows) == 0:
            continue # no need to split

        gain = info_gain(true_rows, false_rows, current_uncertainty)

        if gain >= best_gain:
            best_gain, best_question = gain, question

return best_gain, best_question

```

Builds the tree

"""

get the gain and the question with highest gain
gain, question = find_best_split(rows)

all classes are the same
if gain == 0 **or** depth == 3:
 return Leaf(rows)

true_rows, false_rows = partitions(rows, question)

build the true subtree
true_branch = build_tree(true_rows, depth=depth+1)

build the false subtree
false_branch = build_tree(false_rows, depth=depth+1)

the Question node i.e question to ask at this node
return DecisionNode(question, true_branch, false_branch)

```
print_tree(my_tree)
```

```
Is PetalWidth >= 1.0?
```

```
--> True:
```

```
  Is PetalWidth >= 1.8?
```

```
    --> True:
```

```
      Is PetalLength >= 4.9?
```

```
        --> True:
```

```
          Predict {'Iris-virginica': 31}
```

```
        --> False:
```

```
          Predict {'Iris-virginica': 2, 'Iris-versicolor': 1}
```

```
    --> False:
```

```
      Is PetalLength >= 5.0?
```

```
        --> True:
```

```
          Predict {'Iris-virginica': 2, 'Iris-versicolor': 1}
```

```
        --> False:
```

```
          Predict {'Iris-versicolor': 39}
```

```
--> False:
```

```
  Predict {'Iris-setosa': 39}
```

```
def classify(row, node):  
    """  
    classify a given test data - row  
    the root of decision tree - node  
    """  
  
    if isinstance(node, Leaf):  
        return node.predictions  
  
    if node.question.match(row):  
        return classify(row, node.true_branch)  
    else:  
        return classify(row, node.false_branch)
```

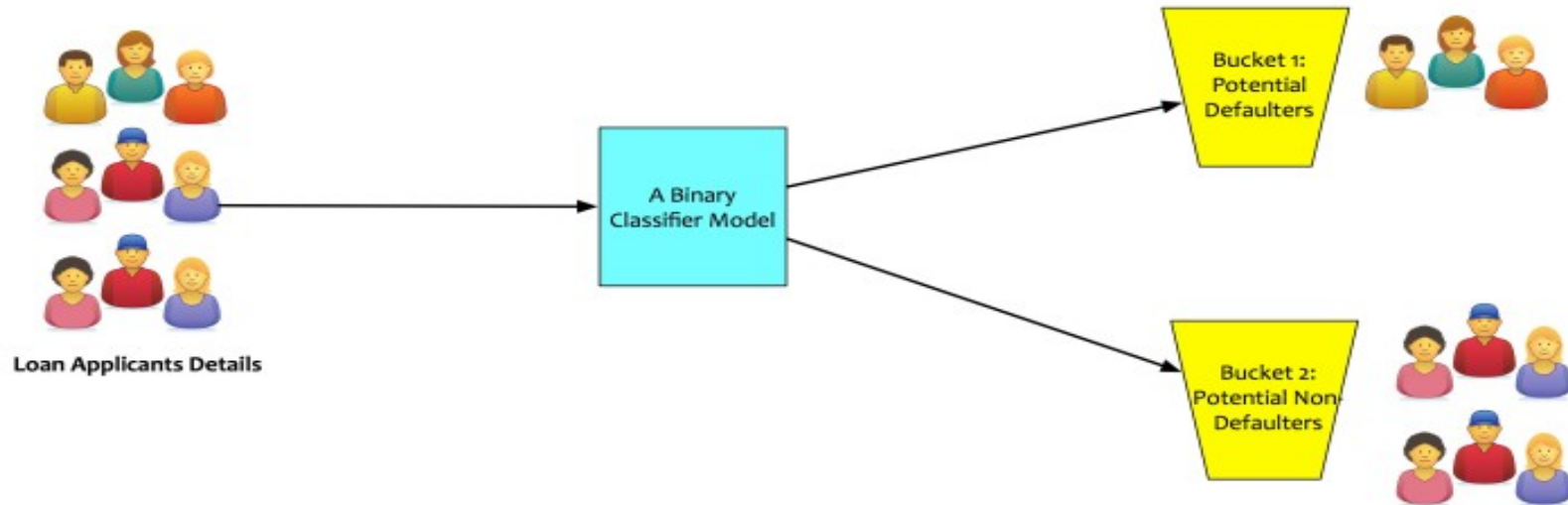
```
classify(test_data[3], my_tree)
```

```
{'Iris-versicolor': 39}
```

Naive Bayes Classifier

072BCT530

Classification



Baye's Theorem

$$\begin{array}{c} \text{Posterior} \\ \text{probability} \end{array} p(A|B) = \frac{\begin{array}{c} \text{Likelihood} \end{array} p(B|A) \begin{array}{c} \text{Prior} \\ \text{probability} \end{array} p(A)}{p(B)}$$

Naive Bayes

- Statistical Classifier
- Predicts class membership probabilities
- Attribute values are independent to each other.

How does it work?

- Find class probabilities of all classes with given feature vector.
- Use Baye's theorem to find each class probability.
- One having highest probability is chosen.

Dataset

Balance Scale Data Set

Download: [Data Folder](#), [Data Set Description](#)

Abstract: Balance scale weight & distance database



Data Set Characteristics:	Multivariate	Number of Instances:	625	Area:	Social
Attribute Characteristics:	Categorical	Number of Attributes:	4	Date Donated	1994-04-22
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	208281

Source:

Generated to model psychological experiments reported by Siegler, R. S. (1976). Three Aspects of Cognitive Development. Cognitive Psychology, 8, 481-520.

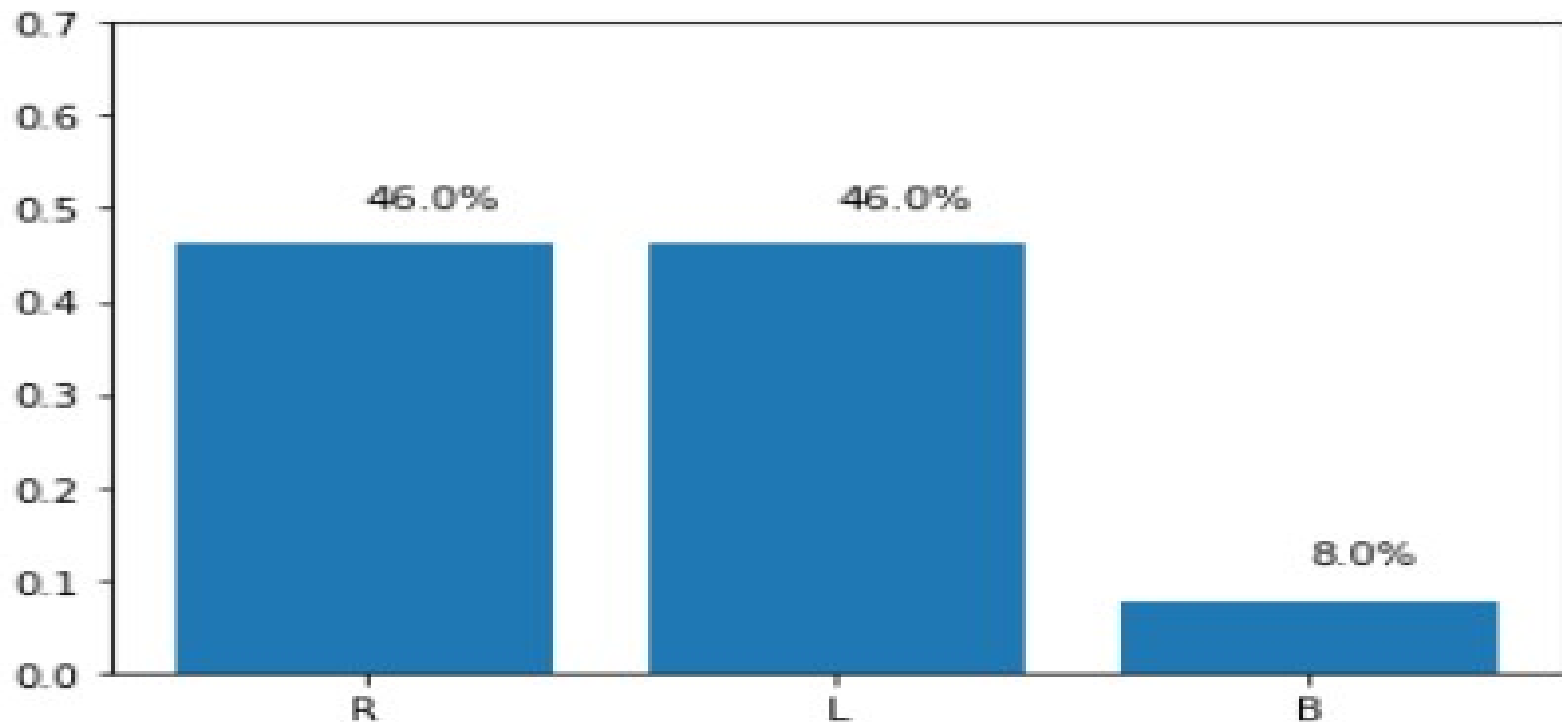
Donor:

Tim Hume (hume '@' ics.uci.edu)

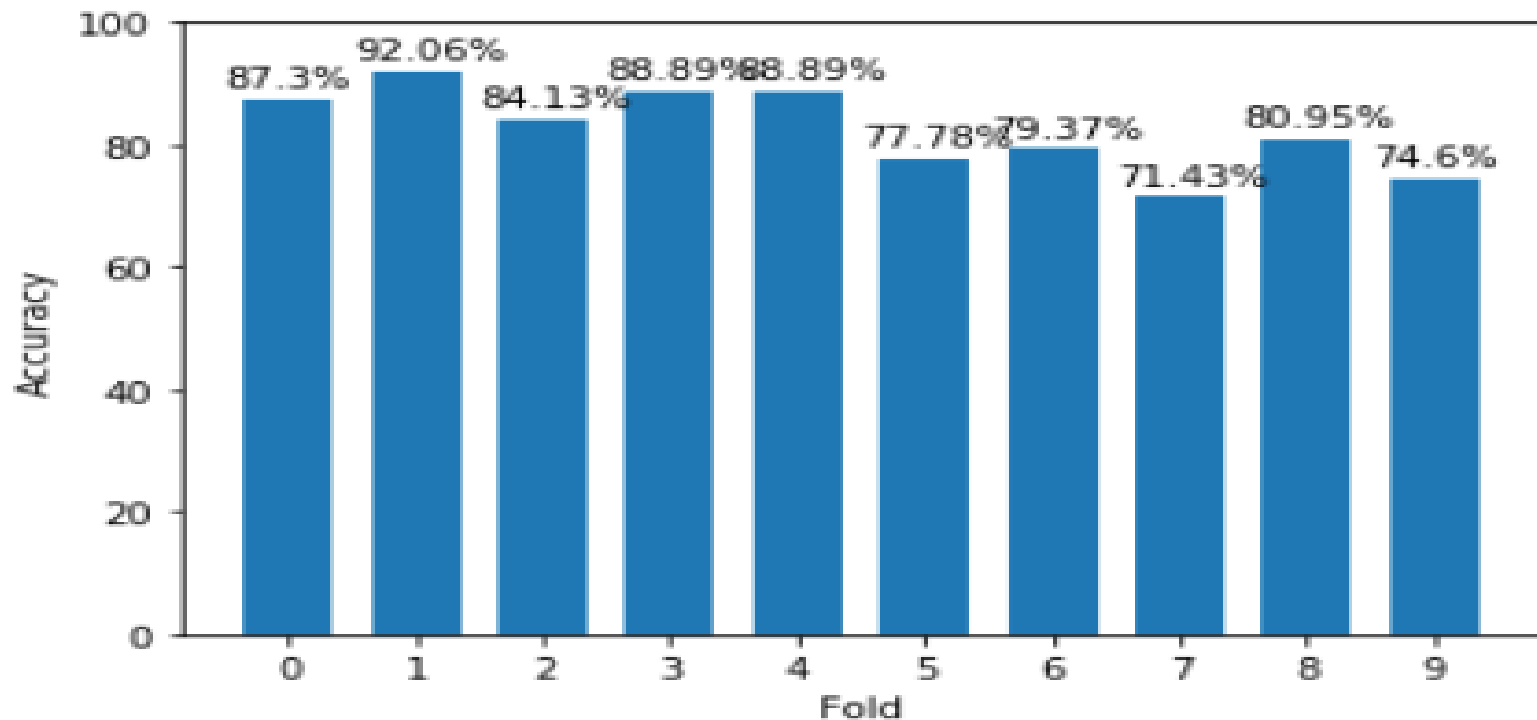
Class Labels

- Right
Left
Balanced

Class Distribution



Cross Validation for 10 folds



Overall Average Accuracy

82.53 %

Thank You