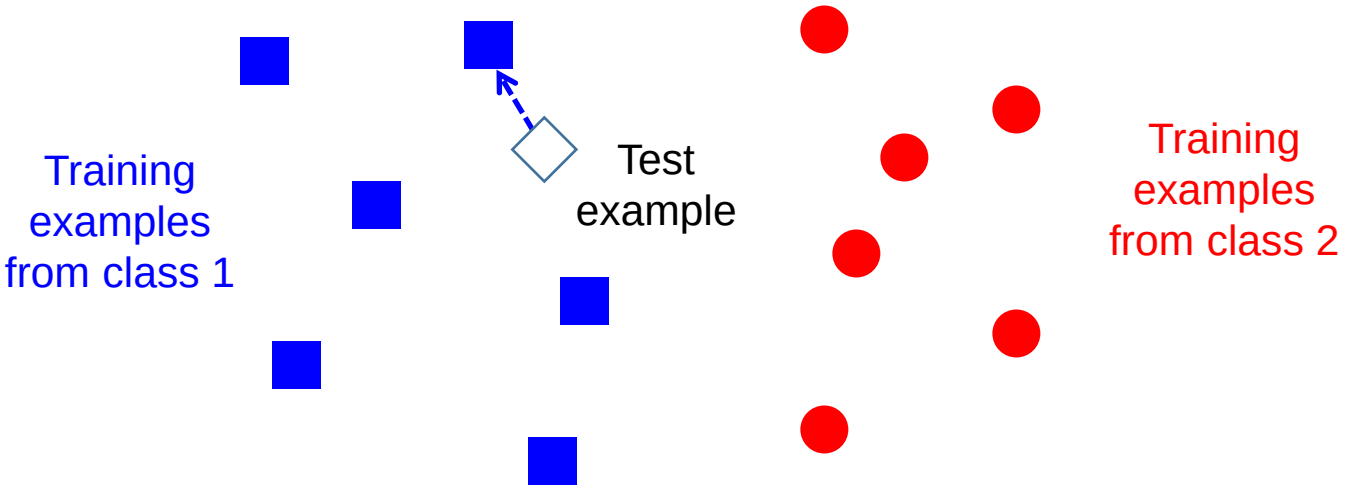
K-Nearest Neighbors and Radius Neighbors Classifiers

Nearest Neighbor Classifier



- All we need is a distance function for our inputs
- No training required!

K-Nearest Neighbors (KNN) Approaches

- We will study the supervised learning approaches KNN classifier and radius neighbors classifier.
- These basic approaches are very simple to implement and they are effective with relatively small data sets. They are slow with large data sets specially when the brute force (exhaustive) search is used.
- They do not have a learning model (no learning parameters) as they require the training data to be available at the prediction time, so they are online learning approaches.
- They can learn complex decision boundaries since they are not constrained to a specific model.

KNN Approaches in sklearn Python Package

- Nearest neighbors' classifiers have two classes in sklearn package:
  1. KNeighborsClassifier() (uses the nearest K number of neighbors for prediction)
  2. RadiusNeighborsClassifier() (uses the nearest neighbors within certain distance)
- To determine the nearest neighbors, a distance measure (Minkowski distance) between $X_1 = [x_1^{(1)}, x_1^{(2)}, \dots, x_1^{(D)}]$ and $X_2 = [x_2^{(1)}, x_2^{(2)}, \dots, x_2^{(D)}]$ vectors in = D-dimensional space, is used:

  p = 1 is the L-1 norm or the Manhattan distance

  p = 2 is the L-2 norm or the Euclidean distance

$$d(X_1, X_2) = \left(\sum_{d=1}^{D} |x_1^{(d)} - x_2^{(d)}|^p\right)^{1/p}$$

- Three searching approaches are offered: brute force, KDTree, and BallTree. The brute force approach is an exhaustive search approach, while KDTree, and BallTree are advanced tree search approaches are used to limit the searching space especially with large multi-dimensional data. Automatic selection of these approaches is an optional parameter.

K-Nearest Neighbors Classifier

Basic Idea

- *k*-NN classification rule is to assign to a test sample the majority category label of its *k* nearest training samples
- In practice, *k* is usually chosen to be odd, so as to avoid ties
- The *k* = 1 rule is generally called the nearest-neighbor classification rule

KNN Neighbor Classifier

- Basic Algorithm (brute force with uniform weights)
  - ✓ Set the number of neighbors K
  - ✓ For each query point x_{query} find its distance, d_i, (in the feature space) with each point x_i in the training data (X = input, y = target labels) as:

$$d_i = distance(x_{query} - x_i)$$

  - ✓ Sort d_i in an ascending order, d_{sorted}. Keep track of indices i.
  - ✓ Find the points in training data that corresponds to the first K elements of $d_{sorted,}$ i.e., find sets $I_K = \{i |\ d_i \in d_{sorted}(1, 2, \ldots K)\}$, $S_K = \{y_i \mid i \in I_k\}$.
  - ✓ find $y_{predicted} = mode(S_K)$; the most frequent label/class.

Data Generation for Classifiers

Using sklearn.metrics.make_classification()

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import classification_report
```

```
from sklearn.datasets import make_classification
```

```
X, y = make_classification(n_samples = 100, n_features=2, n_redundant=0, n_informative=2,
                           n_clusters_per_class=1, n_classes=2, random_state = 0)
```

```
from sklearn.model_selection import train_test_split
```

```
X_train_validation, X_test, y_train_validation,  y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

```
X_train, X_validation, y_train,  y_validation = train_test_split(X_train_validation, y_train_validation,
                                                                 test_size = 0.2,random_state = 0)
```

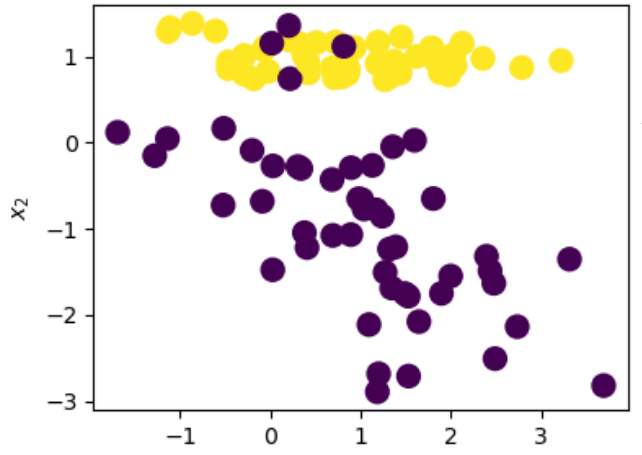
- (X, y) 200 samples of input data and corresponding labels
- 2 features and 2 classes

- (X_test, y_test) is 20% of (X, y) data, (X_train_validation, y_train_validation) 80% of (X, y) data
- (X_validation, y_validation) is 20% of (X_train_validation, y_train_validation),
- (X_train, y_train) is 80% of (X_train_validation, y_train_validation),

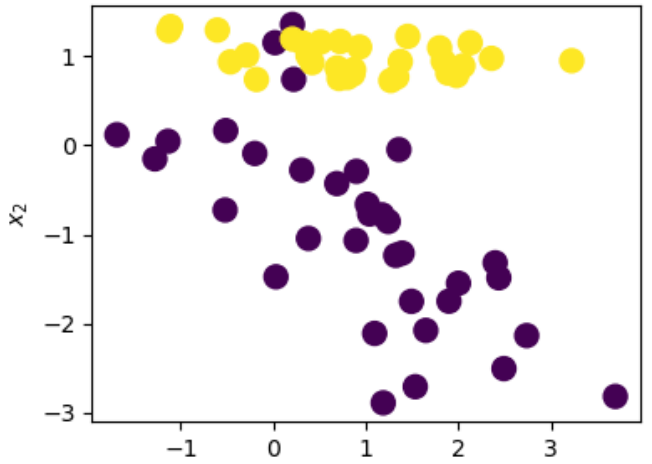
Data Generation for Classifiers

```
plt.figure(1)
plt.subplot(2,2,1)
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.subplot(2,2,2)
plt.scatter(X_train[:, 0], X_train[:, 1], marker='o', c=y_train,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.subplot(2,2,3)
plt.scatter(X_validation[:, 0], X_validation[:, 1], marker='o', c=y_validation,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.subplot(2,2,4)
plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_test,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.show()
```

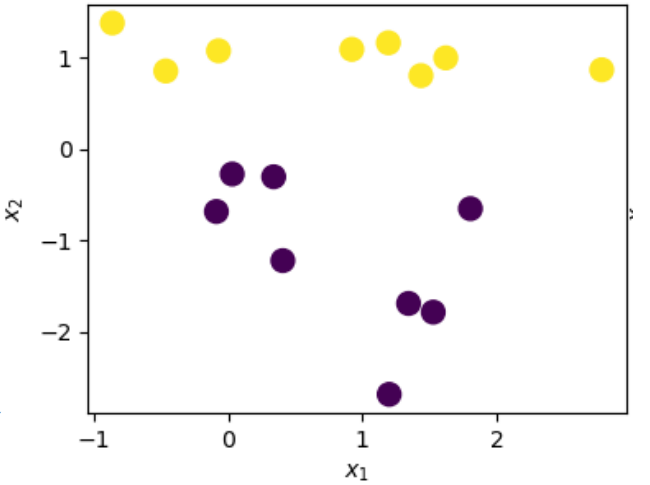
All Data



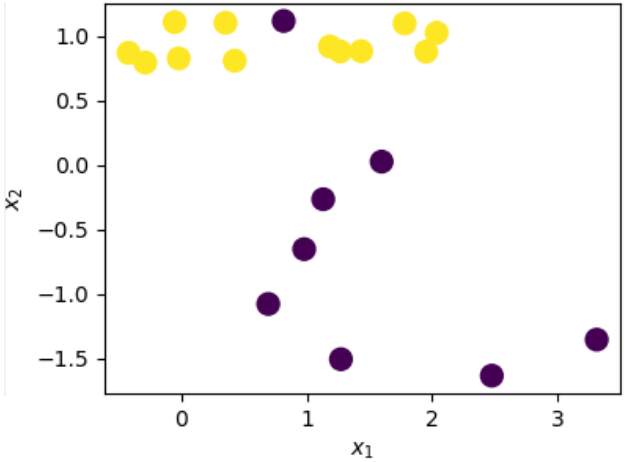
Training Data



Validation Data



Test Data



KNN Classifier Code

```python
from scipy.stats import mode
def knn_trial(X_train, y_train, K, x_test):
    dist = np.zeros(X_train.shape[0])
    for i in range(X_train.shape[0]):
        dist[i] = np.sqrt(np.sum((X_train[i,:] - x_test)**2))
    indx = np.argsort(dist)
    y_K = y_train[indx] # labels sorted based on distance
    # return the most frequent lable (the label of the test point) as scaler.
    return np.int32((mode(y_K[:K,], axis = 0)[0]).item())
```

```python
#evaluate the classifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
y_p = np.zeros(X_test.shape[0])
K_trial = 5
for i in range(X_test.shape[0]):
    y_p[i] = knn_trial(X_train, y_train, K_trial, X_test[i,:]) #find the label of the test point
    print("y test = ", str(y_test[i]), "y predict = ", str(np.int32(y_p[i]))) # print and compare the true and predicted labe
```

```
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  0 y predict =  1
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  1
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  0 y predict =  0
```

KNN Classifier Results

```
print("Confusion Matrix:")
cm = confusion_matrix(y_test, y_p)
print(cm)

Accuracy = (cm[0,0]+ cm[1,1])/ (cm[0,0]+ cm[0,1] + cm[1,0]+ cm[1,1])
print("\nAccuracy: " + str(Accuracy)) # accuracy as a metric

Sensitivity = cm[0,0]/(cm[0,0] + cm[1,0])
print("\nSensitivity: " + str(Sensitivity))

Specificity = cm[1,1]/(cm[1,1] + cm[0,1])
print("\nSpecificity: " + str(Specificity))
```

```
Confusion Matrix:
[[ 6  2]
 [ 0 12]]

Accuracy: 0.9

Sensitivity: 1.0

Specificity: 0.8571428571428571
```

Confusion Matrix:

		True Label	
		Positive	Negative
Predicted Label	Positive	TP	FP
	Negative	FN	TN

Performance Metrics:

sensitivity, recall, hit rate, or true positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

specificity, selectivity or true negative rate (TNR)

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

precision or positive predictive value (PPV)

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} = 1 - \mathrm{FDR}$$

negative predictive value (NPV)

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}} = 1 - \mathrm{FOR}$$

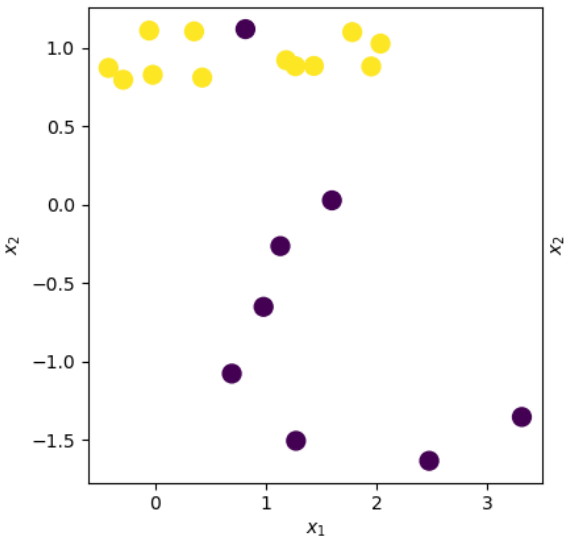
TP: true positive, TN: true negative, FP: false positive, and FN: false negative

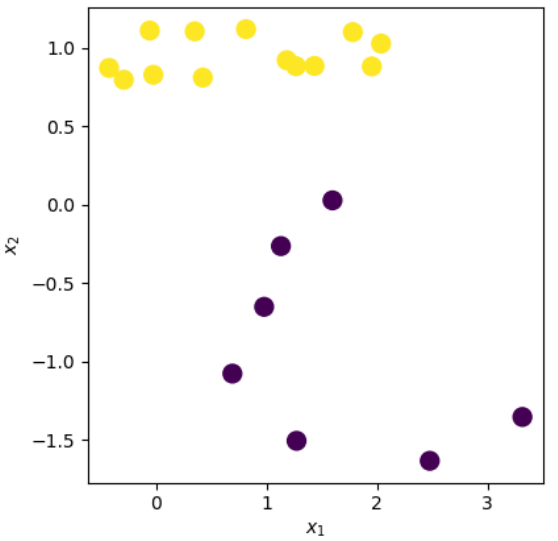
KNN Classifier Tuning with Validation Set

```
from sklearn.neighbors import KNeighborsClassifier
k_scores = np.zeros(10)
for K in range(1,11):
    knn = KNeighborsClassifier(n_neighbors = K)
    knn.fit(X_train,y_train)
    k_scores[K-1] = knn.score(X_validation,y_validation)
    print(str(K) + " Neighbor(s) score", str(k_scores[K-1]))
```

```
1 Neighbor(s) score 0.9375
2 Neighbor(s) score 0.9375
3 Neighbor(s) score 1.0
4 Neighbor(s) score 1.0
5 Neighbor(s) score 1.0
6 Neighbor(s) score 1.0
7 Neighbor(s) score 1.0
8 Neighbor(s) score 1.0
9 Neighbor(s) score 1.0
10 Neighbor(s) score 1.0
```

True/Test





Predicted

```
K_best = np.argmax(k_scores)
print("Best K = " + str(K_best + 1))
knn_test = KNeighborsClassifier(n_neighbors = K_best + 1)
knn_test.fit(X_train,y_train)
knn_accuracy= knn_test.score(X_test,y_test)
y_predict = knn_test.predict(X_test)
plt.figure(2)
plt.subplot(1,2,1)
plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_test,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.subplot(1,2,2)
plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_predict,s=100)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.show()
```

```
Best K = 3
```

```
knn_accuracy
```

```
0.95
```

Radius Neighbors Classifier

Radius Neighbor Classifier

- It is usually used when the data is sparse.

- Basic Algorithm (brute force with uniform weights)
  - ✓ Set the Radius value R
  - ✓ For each query point x_{query} find its distance, d_i, (in the feature space) with each point x_i in the training data as:

$$d_i = distance(x_{query} - x_i)$$

  - ✓ Find the points in training data that have $d_i \leq R$. i.e. find set $S_R = \{y_i | \ d_i \leq R\}$
  - ✓ find $y_{predicted} = mode(S_R)$; the most frequent label/class.

Radius Neighbor Classifier Code

```
from scipy.stats import mode
def rad_neighbors(X_train, y_train, R, x_test, most_frequent):
    within_distance = []
    for i in range(X_train.shape[0]):
        dist = np.sqrt(np.sum((X_train[i,:] - x_test)**2))
        if dist <= R:
            within_distance.append(y_train[i])
    if within_distance:
        return np.int32((mode(within_distance)[0]).item())
    else:
        return most_frequent
```

```
#evaluate the classifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
y_p = np.zeros(X_test.shape[0])
R_trial = 0.2
most_frequent = np.int32((mode(y_train)[0].item()))
print(most_frequent)
for i in range(X_test.shape[0]):
    y_p[i] = rad_neighbors(X_train, y_train, R_trial, X_test[i,:], most_frequent) #find the label of the test point
    print("y test = ", str(y_test[i]), "y predict = ", str(np.int32(y_p[i]))) # print and compare the true and predicted labe

print("Confusion Matrix:")
cm = confusion_matrix(y_test, y_p)
print(cm)

Accuracy = (cm[0,0]+ cm[1,1])/ (cm[0,0]+ cm[0,1] + cm[1,0]+ cm[1,1])
print("\nAccuracy: " + str(Accuracy)) # accuracy as a metric
Sensitivity = cm[0,0]/(cm[0,0] + cm[1,0])
print("\nSensitivity: " + str(Sensitivity))
Specificity = cm[1,1]/(cm[1,1] + cm[0,1])
print("\nSpecificity: " + str(Specificity))
```

Radius Neighbor Classifier Results

```
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  0
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  1
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  1 y predict =  1
y test =  1 y predict =  1
y test =  0 y predict =  0
y test =  0 y predict =  0
Confusion Matrix:
[[ 7  1]
 [ 1 11]]

Accuracy: 0.9

Sensitivity: 0.875

Specificity: 0.9166666666666666
```

Confusion Matrix:

		True Label	
		Positive	Negative
Predicted Label	Positive	TP	FP
	Negative	FN	TN

Performance Metrics:

sensitivity, recall, hit rate, or true positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

specificity, selectivity or true negative rate (TNR)

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

precision or positive predictive value (PPV)

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} = 1 - \mathrm{FDR}$$

negative predictive value (NPV)

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}} = 1 - \mathrm{FOR}$$

TP: true positive, TN: true negative, FP: false positive, and FN: false negative

Radius Neighbor Tuning with Validation Set

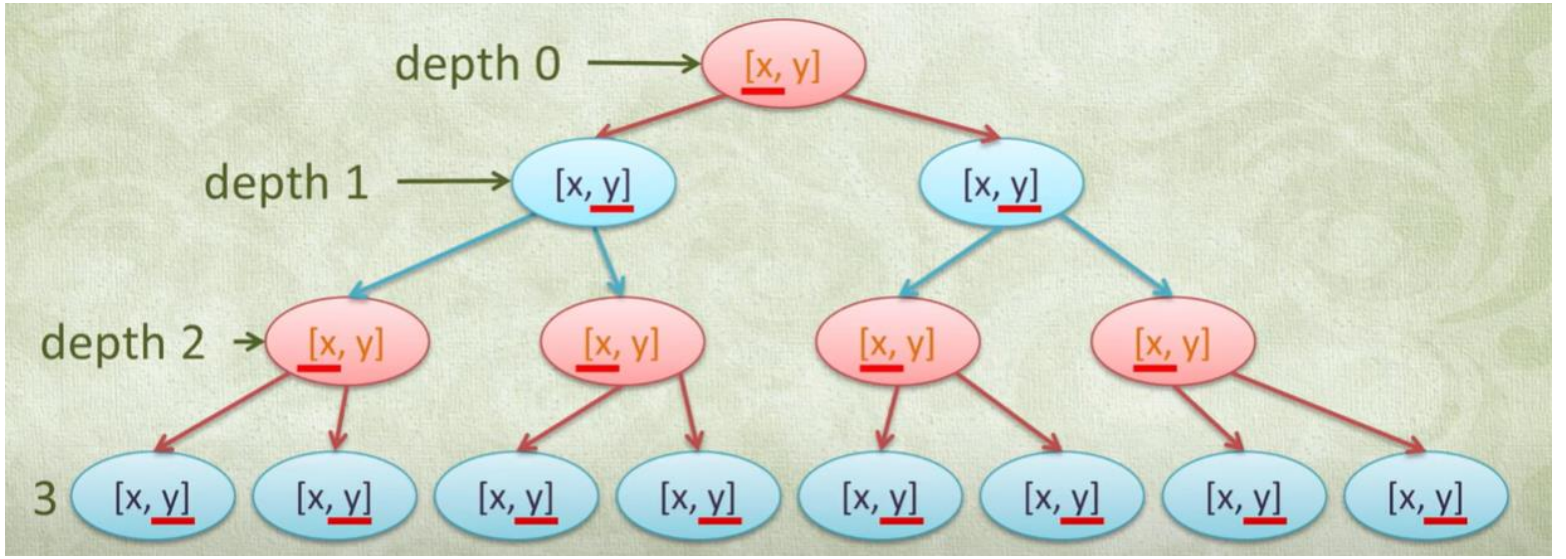
```
from sklearn.neighbors import RadiusNeighborsClassifier
k_scores = np.zeros(10)
R = np.linspace(0.05,0.7,10)
for i in range(len(R)):
    knn = RadiusNeighborsClassifier(radius = R[i], outlier_label = 'most_frequent')
    knn.fit(X_train,y_train)
    k_scores[i] = knn.score(X_validation,y_validation)
    print(str(R[i]) + ": score", str(k_scores[i]))
R_best = R[np.argmax(k_scores)]
print("Best R = " + str(R_best))
knn_test = RadiusNeighborsClassifier(radius = R_best, outlier_label = 'most_frequent')
knn_test.fit(X_train,y_train)
knn_accuracy= knn_test.score(X_test,y_test)
y_predict = knn_test.predict(X_test)
print("Accuracy = " + str(knn_accuracy))
```

```
0.05: score 0.5625
0.12222222222222222: score 0.6875
0.19444444444444442: score 0.6875
0.26666666666666666: score 0.875
0.33888888888888885: score 0.9375
0.41111111111111104: score 0.9375
0.4833333333333333: score 1.0
0.5555555555555556: score 1.0
0.6277777777777778: score 1.0
0.7: score 1.0
Best R = 0.4833333333333333
Accuracy = 0.95
```

KD Tree Nearest Neighbor

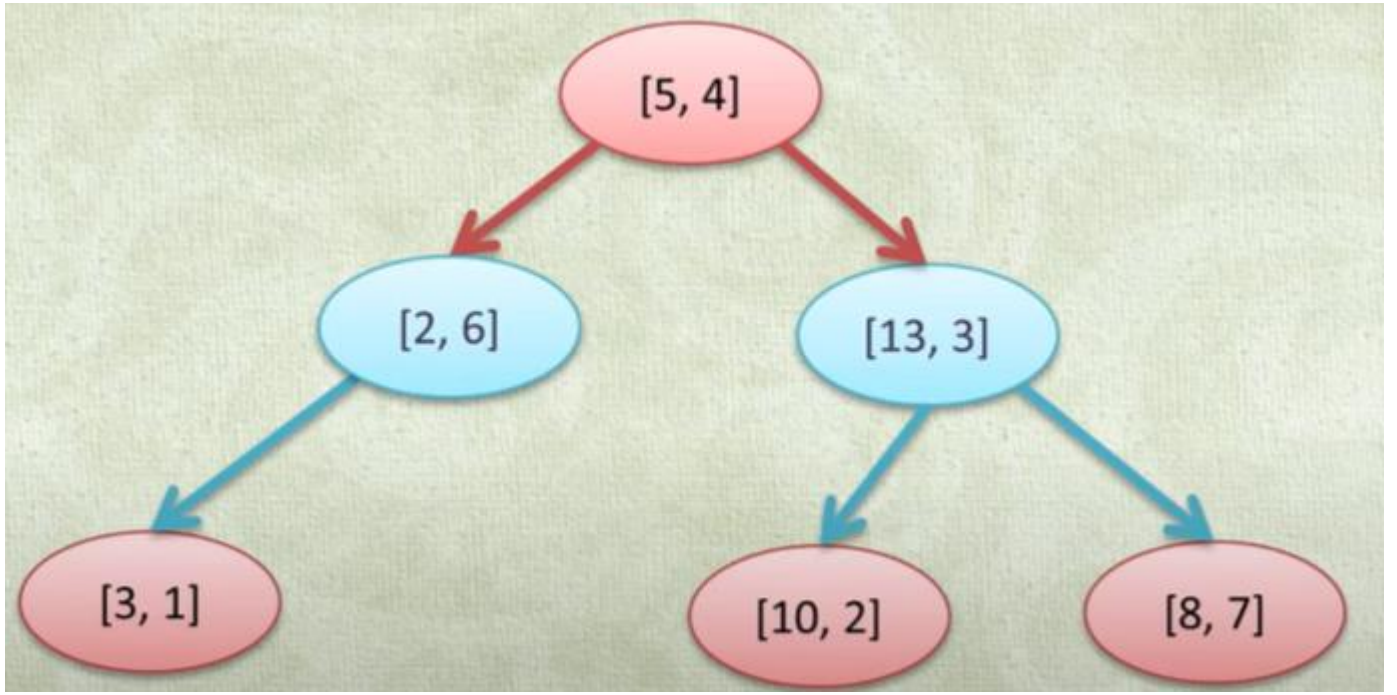
KD Tree

- a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space.

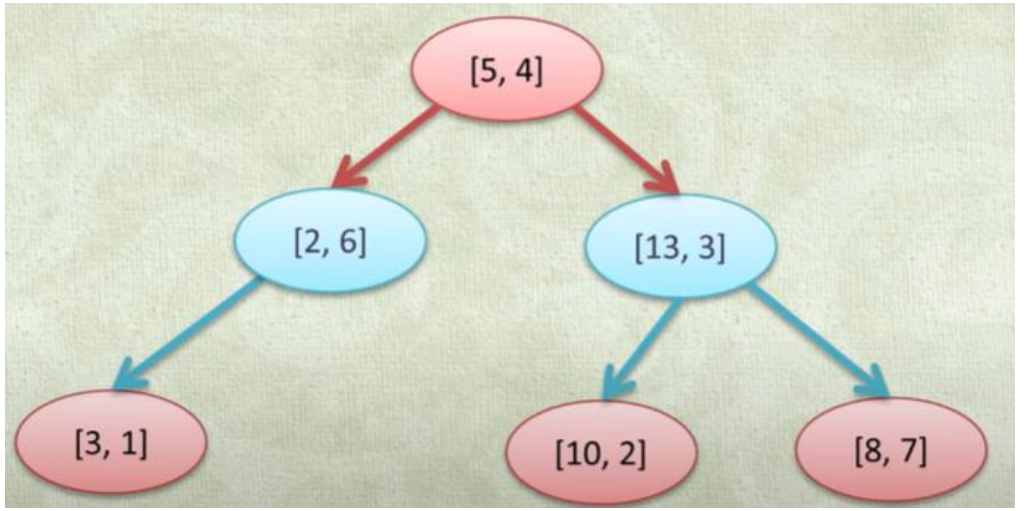


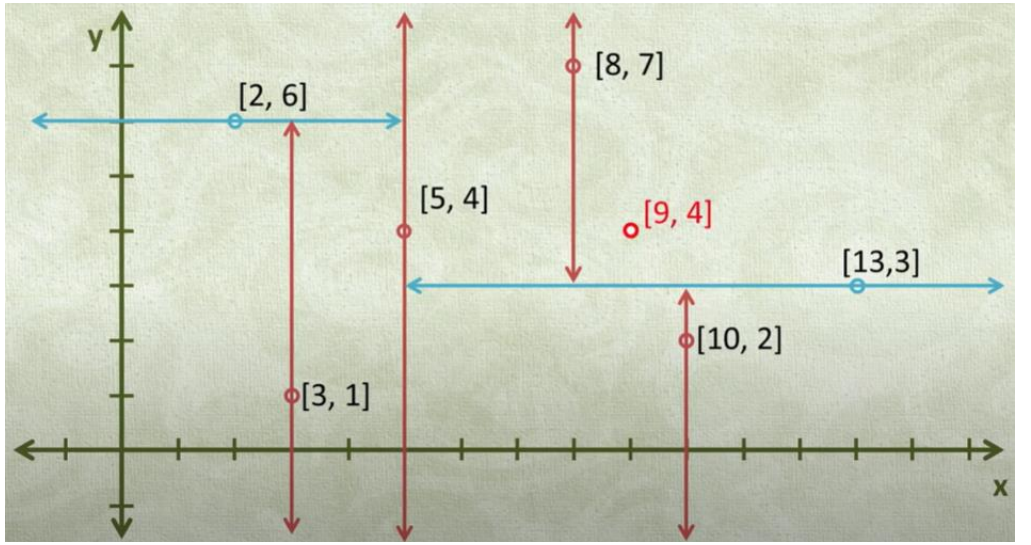
KD Construction

[5, 4] [2, 6] [13,3] [8, 7] [3, 1] [10, 2]

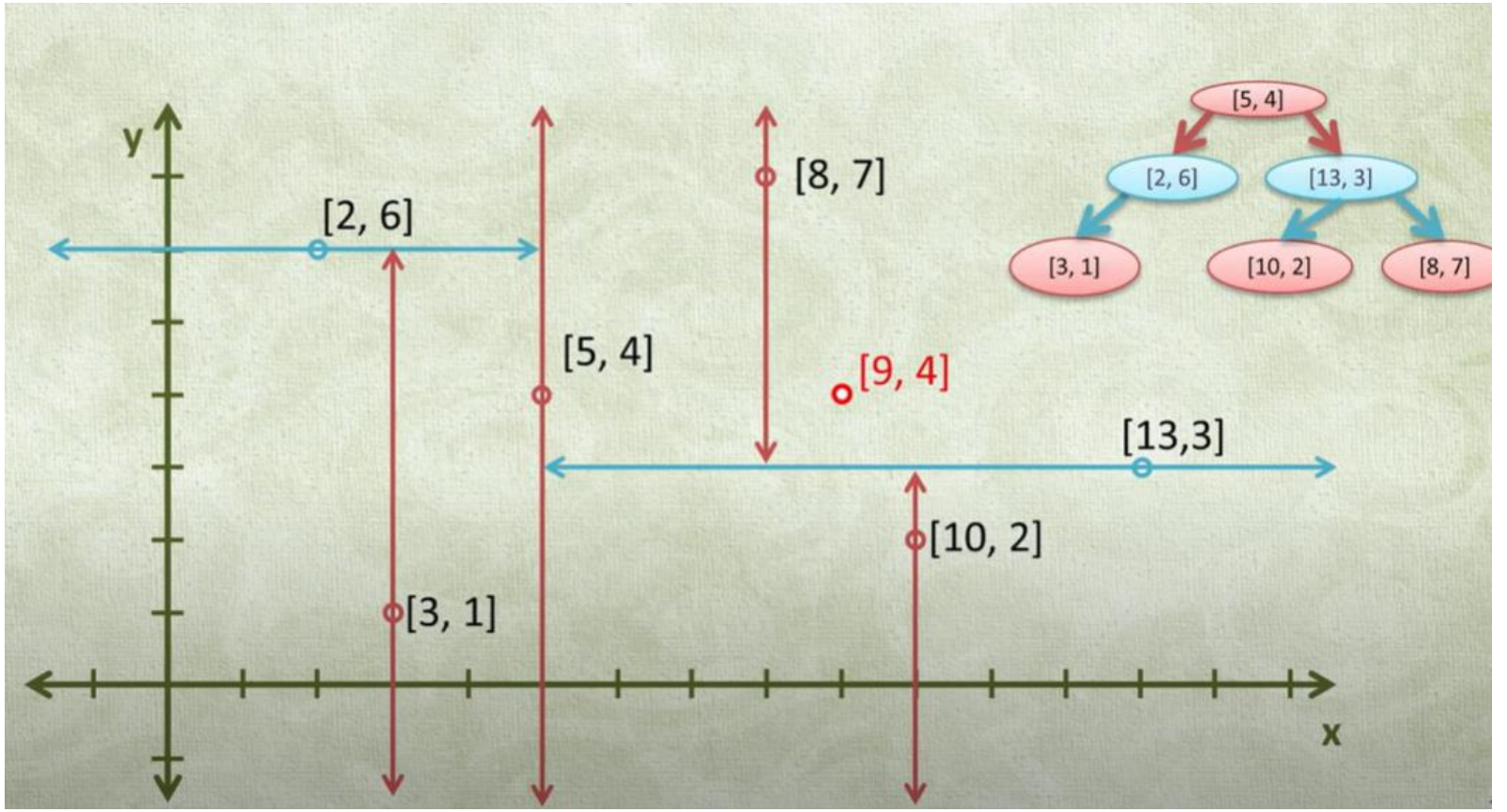


KD Tree

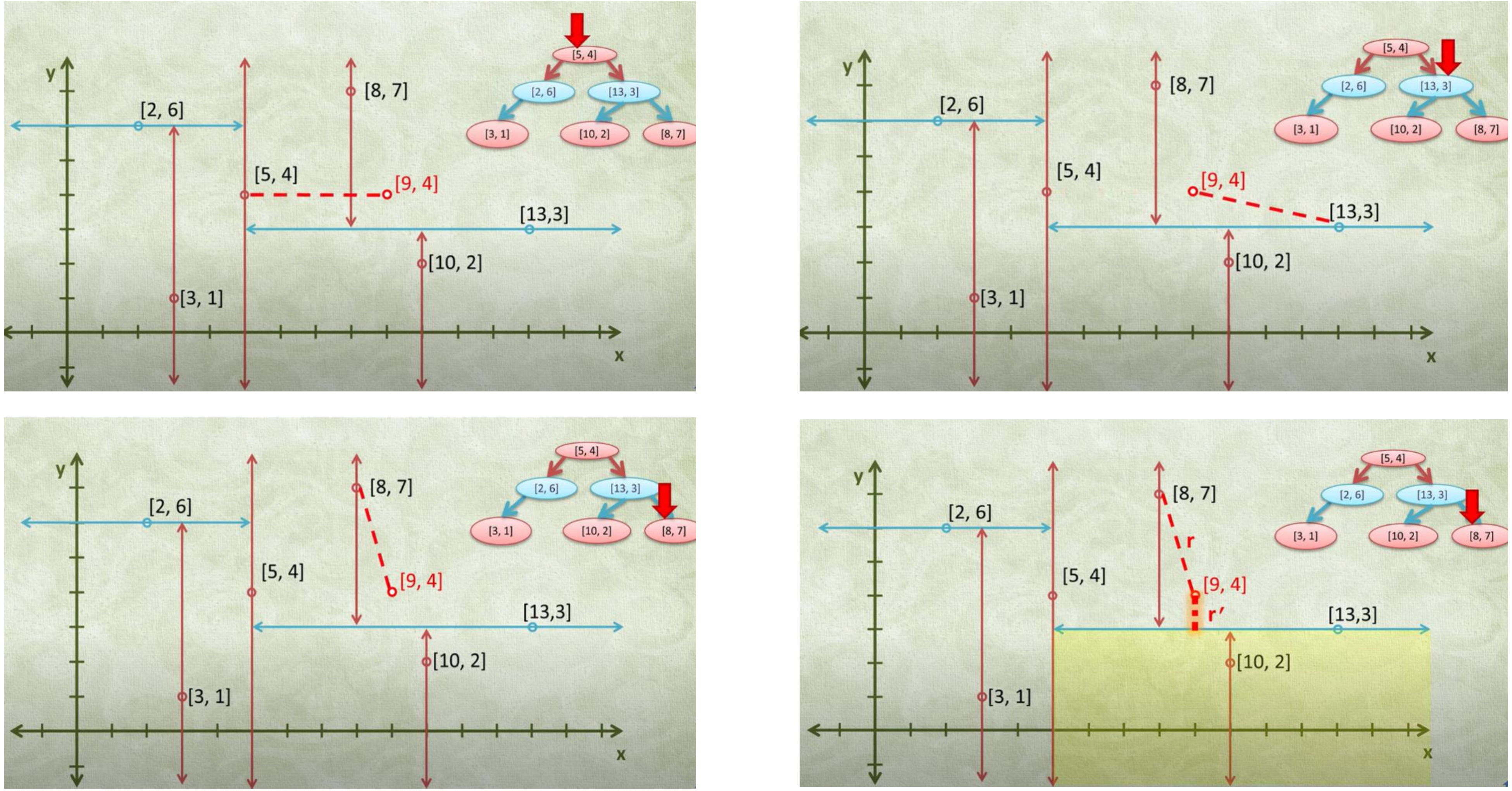




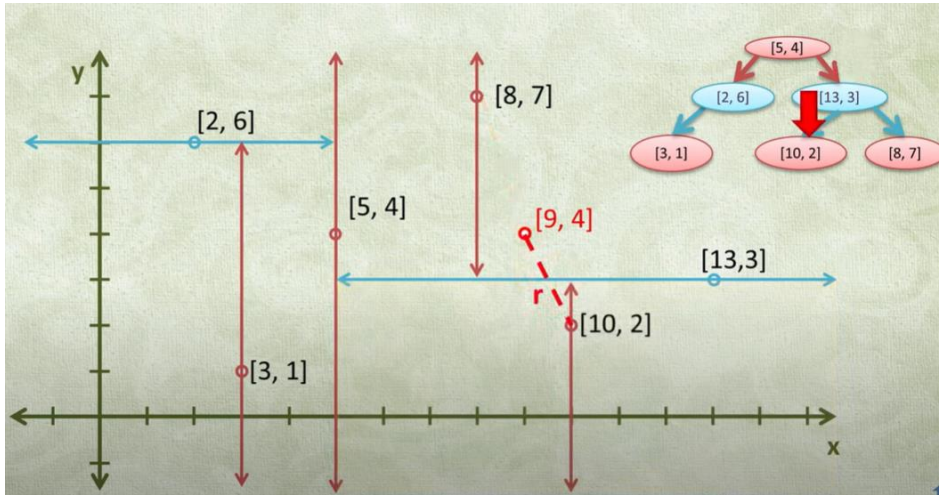
KD Tree: Finding Nearest Neighbor



KD Tree: Finding Nearest Neighbor



KD Tree: Finding Nearest Neighbor

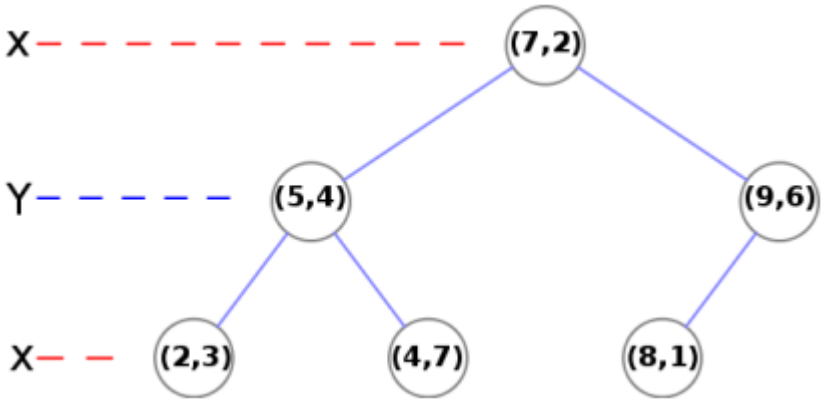


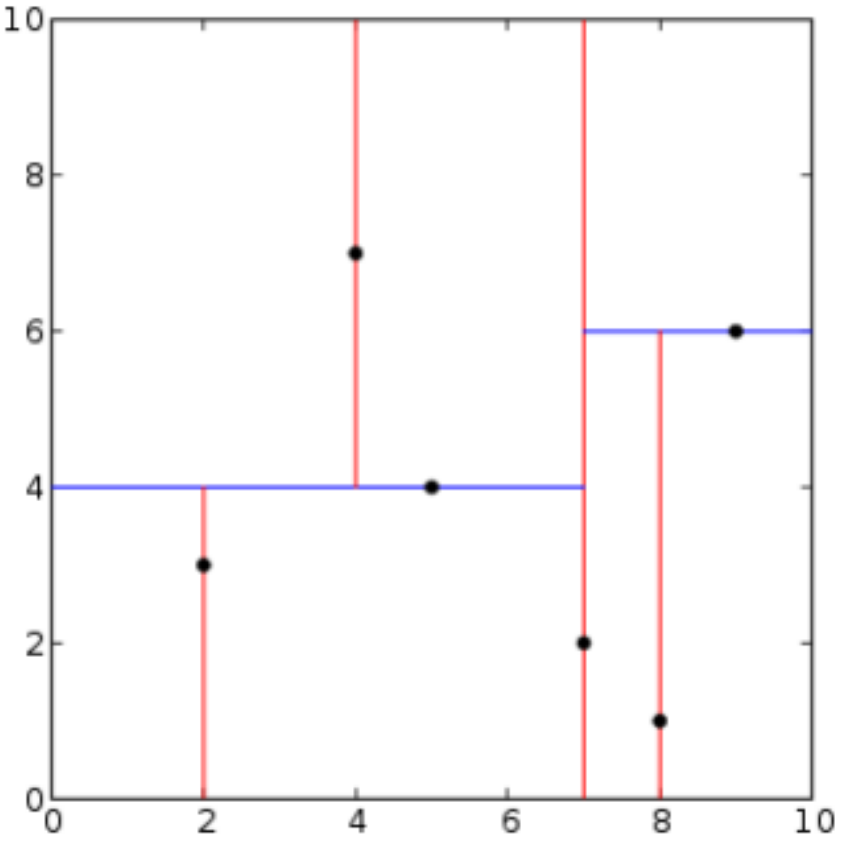
Practice

- Construct the KD Tree for the following Data and Draw the feature Space divided accordingly:

  [7, 2], [9, 6], [5,4], [2,3], [8,1], [4,7]

Solution





Thank You!