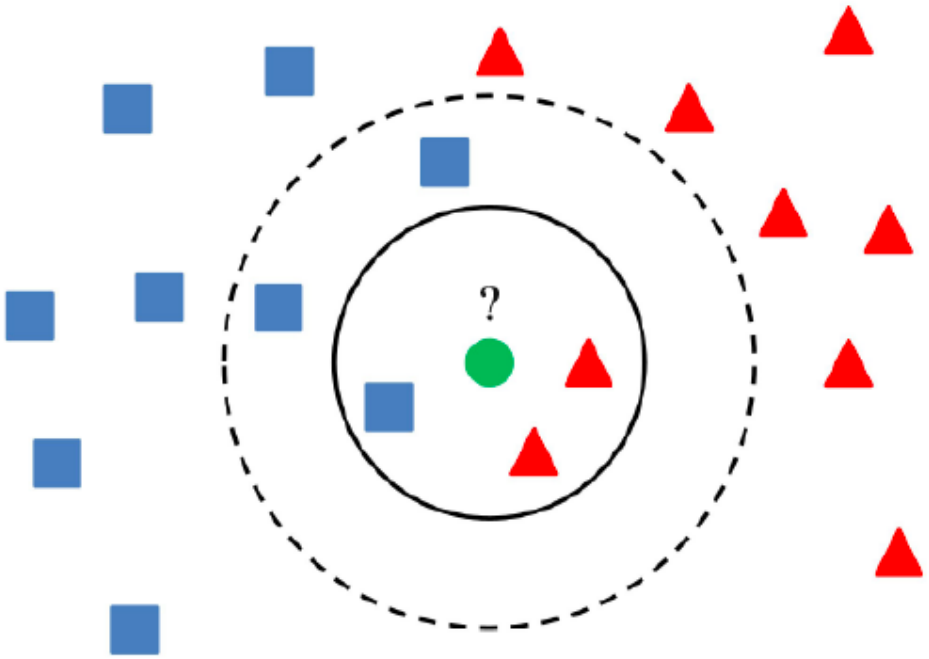
## K-Nearest Neighbors Algorithm




## Nearest Neighbors Classification

Neighbors-based classification is a type of instance-based learning or non-generalizing learning.

It does not attempt to construct **a general internal model**, but **simply stores instances of the training data**.

**Classification** is computed from a **simple majority vote of the nearest neighbors of each point**: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

scikit-learn implements two different nearest neighbors classifiers:

**KNeighborsClassifier** implements learning based on the **nearest neighbors of each query point**, where is an **integer value specified by the user**.

**RadiusNeighborsClassifier** implements learning based on the number of neighbors within a fixed **radius** of each training point, where is a floating-point value specified by the user.

The K-neighbors classification in KNeighborsClassifier is the most commonly used technique. The optimal choice of the value is **highly data-dependent**: in general a **larger suppresses the effects of noise**, b**ut makes the classification boundaries less distinct.**

In cases where the **data is not uniformly sampled**, **radius-based neighbors classification** in **RadiusNeighborsClassifier** can be a better choice.

The user specifies a **fixed radius r** , such that points in sparser neighborhoods use fewer nearest neighbors for the classification. For high-dimensional parameter spaces, this method becomes less effective due to the so-called **"curse of dimensionality"**.

The **basic nearest neighbors classification** uses **uniform weights**: that is, the value assigned to a **query point** is computed **from a simple majority vote of the nearest neighbors**. Under some circumstances, it is better to weight the neighbors such that nearer neighbors contribute more to the fit.

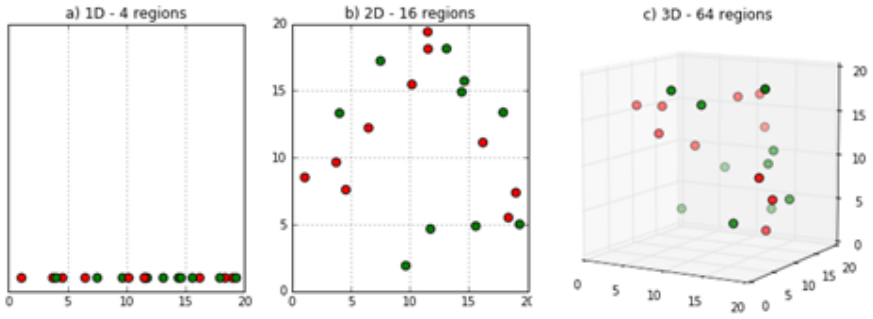
## What is the curse of dimensionality?

Credit : https://deepai.org/machine-learning-glossary-and-terms/curse-of-dimensionality

The curse of dimensionality refers to the phenomena that occur when classifying, organizing, and analyzing high dimensional data that does not occur in low dimensional spaces, specifically the issue of data sparsity and "closeness" of data.

## Issues

**Sparsity of data occurs** when moving to higher dimensions. The volume of the space represented grows so quickly that the data cannot keep up and thus becomes sparse, as seen below. The sparsity issue is a major one for anyone whose goal has some statistical significance.




As the **data space *seen above *moves from one dimension to two dimensions and finally to three dimensions**, the given **data fills less and less of the data space**. In order to maintain an accurate representation of the space, the data for analysis grows exponentially.

The second issue that arises is related to sorting or classifying the data.

In low dimensional spaces, data may seem very similar but the higher the dimension the further these data points may seem to be.

The two wind turbines below seem very close to each other in two dimensions but separate when

## Mitigating Curse of Dimensionality

To mitigate the problems associated with high dimensional data a suite of techniques generally referred to 'Dimensionality reduction techniques' are used. Dimensionality reduction techniques fall into one of the two categories.

```
1.Feature selection
    a.Low Variance filter
    b.High Correlation filter
    c.Multicollinearity
    d.Feature Ranking
    e.Forward selection

2.Feature extraction
    a.Principal Component Analysis (PCA)
    b.Factor Analysis (FA)
    c.Independent Component Analysis (ICA)
```

## Nearest Neighbors Regression

**Neighbors-based regression** can be used in cases where the data labels are **continuous rather than discrete variables**.

The label assigned to a query point is computed based on the **mean** of the labels of its nearest neighbors.

scikit-learn implements **two different neighbors regressors**:

**KNeighborsRegressor** implements learning based on the nearest neighbors of each query point, where is an integer value specified by the user.

**RadiusNeighborsRegressor** implements learning based on the neighbors within a fixed radius of the query point, where is a floating-point value specified by the user.

The **basic nearest neighbors regression** uses uniform weights: that is, each point in the local neighborhood contributes **uniformly to the classification of a query point**.

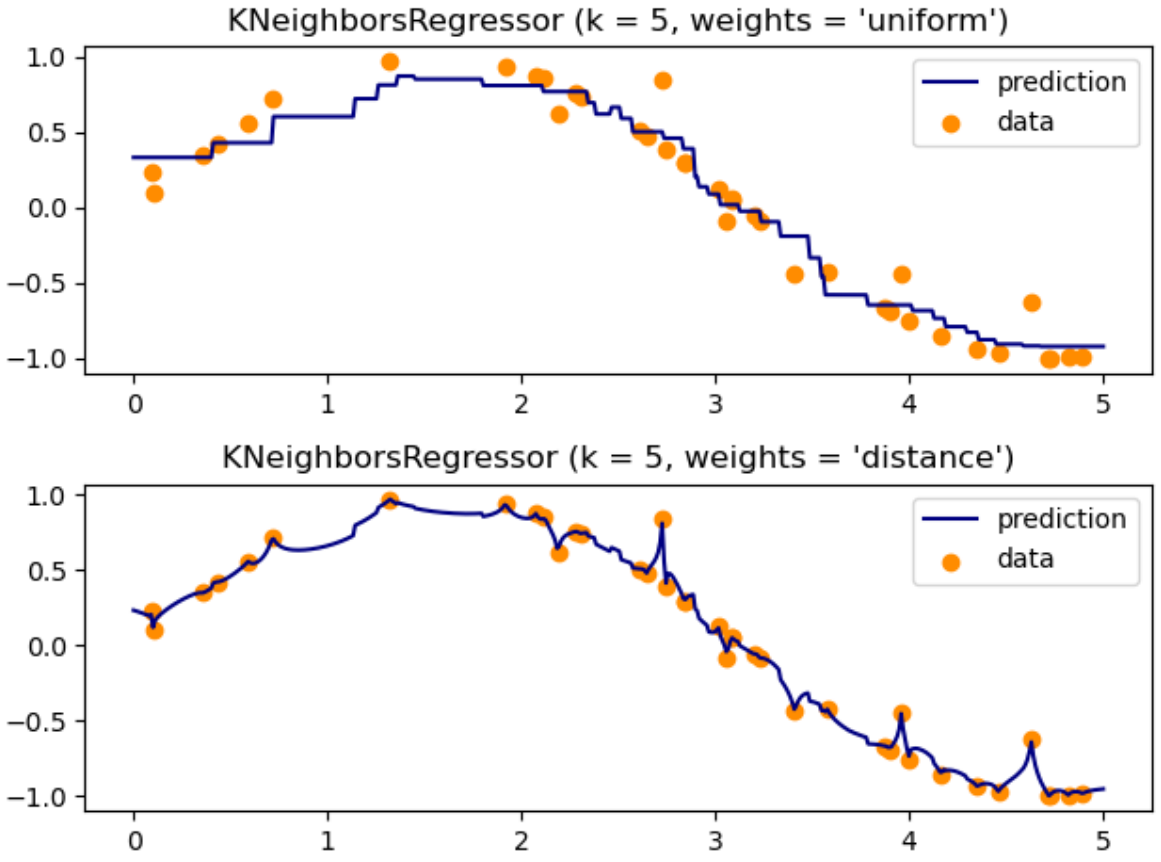
Under some circumstances, it can be **advantageous** to **weight points such that nearby points contribute more to the regression than faraway points**.

This can be accomplished through the weights keyword.

The default value, **weights = 'uniform',assigns equal weights to all points.**

**weights = 'distance'assigns weights proportional to the inverse of the distance from the query point**.

**Alternatively**, a **user-defined function of the distance can be supplied, which will be used to compute the weights.**




# Distance Metrics

credits : https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html

For the **algorithm to work best on a particular dataset** we need to choose the most **appropriate distance metric** accordingly.

There are a lot of **different distance metrics available**, but we are only going to talk about a few widely used ones.

## Minkowski Distance

It is a **metric intended** for **real-valued vector spaces**. We can calculate Minkowski distance only in a normed vector space, which means in a space where distances can be represented as a vector that has a length and the lengths cannot be negative.

There are a **few conditions** that the **distance metric must satisfy**:

1.Non-negativity: d(x, y) >= 0

2.Identity: d(x, y) = 0 if and only if x == y

3.Symmetry: d(x, y) = d(y, x)

4.Triangle Inequality: d(x, y) + d(y, z) >= d(x, z)

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

This above formula for Minkowski distance is in generalized form and we can manipulate it to get different distance metrices.

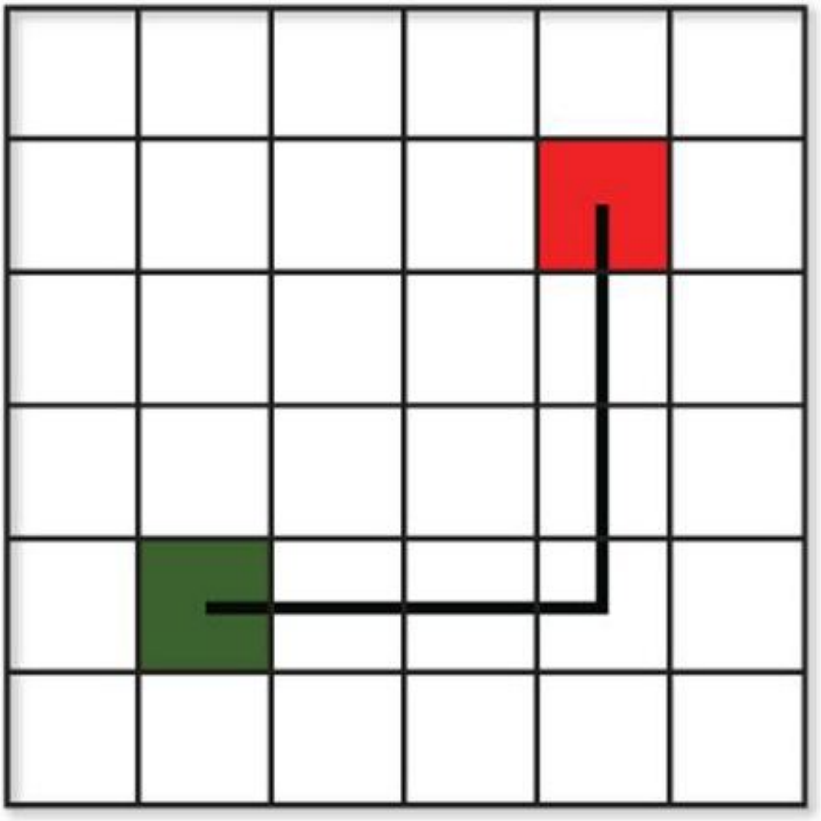
The p value in the formula can be manipulated to give us different distances like:

**p = 1**, when p is set to 1 we get **Manhattan distance**

**p = 2**, when p is set to 2 we get **Euclidean distance**

## Manhattan Distance

This distance is also known as taxicab distance or city block distance, that is because the way this distance is calculated. The distance between two points is the **sum of the absolute differences of their Cartesian coordinates**.



Manhattan Distance

As we know we get the formula for Manhattan distance by substituting p=1 in the Minkowski distance formula.

$$d= \sum_{i=1}^{n} |x_i - y_i|$$

Suppose we have two points as shown in the image the red(4,4) and the green(1,1).

And now we have to calculate the distance using Manhattan distance metric.

We will get,

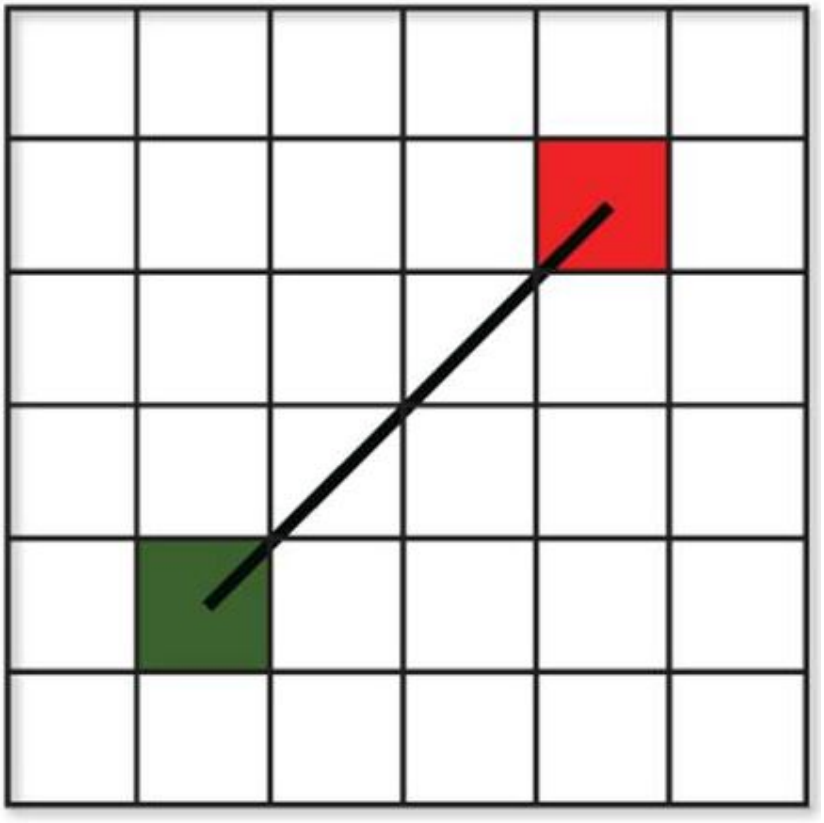
```
d=(4-1)+(4-1)=6
```

This **distance** is preferred over **Euclidean distance** when we have a case of **high dimensionality.**

## Euclidean Distance

This **distance** is the most widely used one as it is the default metric that SKlearn library of Python uses for **K-Nearest Neighbour**.

It is a **measure of the true straight** line **distance between two points** in **Euclidean space**.



Euclidean Distance

It can be used by setting the value of **p equal to 2** in **Minkowski distance metric.**

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Now suppose we have two point the red (4,4) and the green (1,1).

And now we have to calculate the distance using Euclidean distance metric.

We will get,

```
√((4-1)*2+(4-1)*2)

= 4.24
```

## Cosine Distance

This **distance metric** is used mainly to **calculate similarity between two vectors**. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in the same direction.

It is often used to measure **document similarity in text analysis**.

When used with KNN this distance gives us a new perspective to a business problem and lets us find some hidden information in the data which we didn't see using the above two distance matrices.

It is also used in text analytics to find similarities between two documents by the number of times a particular set of words appear in it.

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|}$$

Using this formula we will get a **value** which tells us about the **similarity between the two vectors** and **1-cosΘ** will give us their **cosine distance**.

Using this distance we get values **between 0 and 1**, where **0 means** the vectors are 100% similar to each other and 1 means they are not similar at all.

## Jaccard Distance

The Jaccard coefficient is a similar method of comparison to the Cosine Similarity due to how both methods compare one type of attribute distributed among all data.

The Jaccard approach looks at the two data sets and finds the incident where both values are equal to 1. So the resulting value reflects how many 1 to 1 matches occur in comparison to the total number of data points.

This is also known as the frequency that 1 to 1 match, which is what the Cosine Similarity looks for, how frequent a certain attribute occurs.

It is extremely sensitive to small samples sizes and may give erroneous results, especially with very small data sets with missing observations.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

## Hamming Distance

Hamming distance is a metric for comparing two binary data strings.

While comparing two binary strings of equal length, Hamming distance is the number of bit positions in which the two bits are different.

The Hamming distance method looks at the whole data and finds when data points are similar and dissimilar one to one.

The Hamming distance gives the result of how many attributes were different.

This is used mostly when you **one-hot encode** your data and need to find **distances between the two binary vectors.**

## Step by Step Procedure of KNN Algorithm

```
1.Load the data

2.Initialize K to your chosen number of neighbors

3. For each example in the data

      3.1 Calculate the distance between the query example and the
          current example from the data.

      3.2 Add the distance and the index of the example to an ordered
          collection.

4. Sort the ordered collection of distances and indices from smallest to
   largest (in ascending order) by the distances.


5.Pick the first K entries from the sorted collection.
```

```
6.Get the labels of the selected K entries.

7.If** regression**, return **the mean of the K labels**.

8.If **classification**, return **the mode of the K labels**.
```

## Choosing the right value for K

To select **the K** that's right for your data, we run the **KNN algorithm several times with different values of K** and choose the K that reduces the **number of errors** we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

1.As we decrease the value of K to 1, our predictions become less stable

2.Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point). Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.

3.In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

## Advantages

1.The algorithm is **simple** and **easy** to implement.

2.There's no need to build a model, **tune several parameters**, or make **additional assumptions**.

3.The algorithm is **versatile**. It can be used for **classification, regression, and search.**

## Disadvantages

The **algorithm** gets significantly **slower** as the number of examples and/or **predictors/independent variables increase**.

## Nearest Neighbor Algorithms

## Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset.

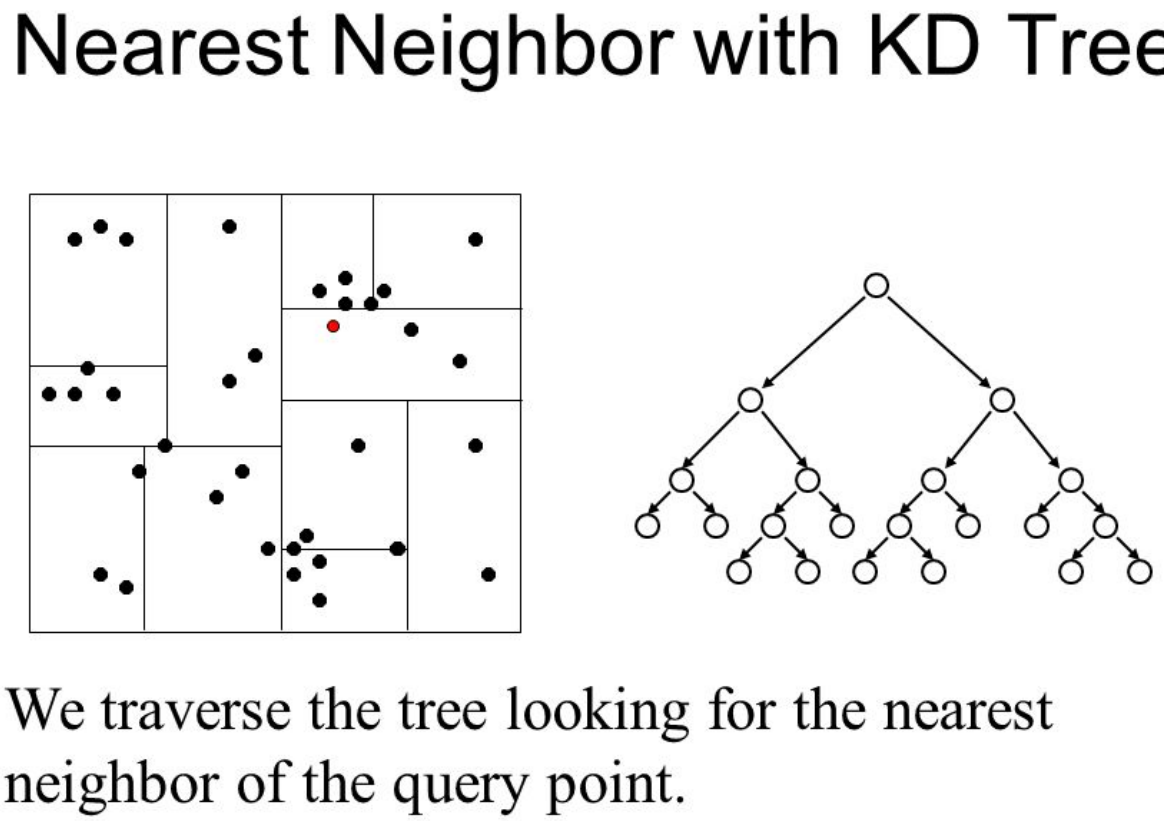
The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset:

for N samples in D dimensions, this approach scales as O[DN**2].

Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples grows, the brute-force approach quickly becomes infeasible

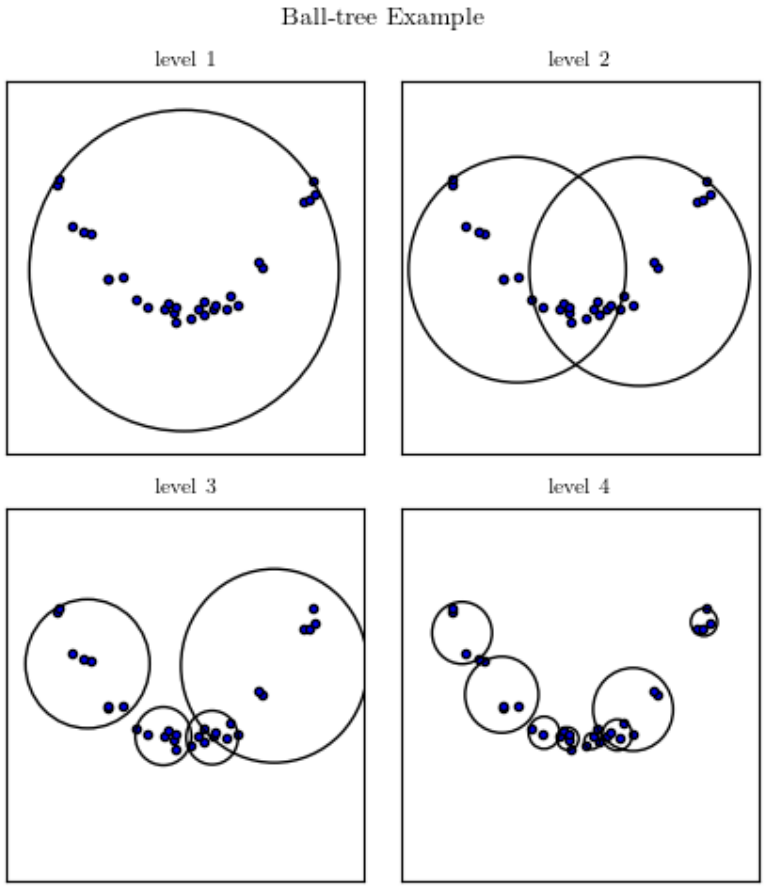
## K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample.




## Ball Tree

To address the inefficiencies of KD Trees in higher dimensions, the ball tree data structure was developed. Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions.




## Hyperparameters in KNN

1.Number of leaf_size

2.n_neighbors=odd value

3.Weights=uniform, distance

4.Metric=minkowski,Euclidean

## Assumptions

1. The data is in feature space, which means data in feature space can be measured by **distance metrics** such as **Manhattan, Euclidean, etc.**

2. Each of the training data points consists of a set of vectors and a class label associated with each vector.

3. Desired to have **'K' as an odd number** in case of 2 class classification.