Combining multiple learners: Model combination schemes, Voting, Ensemble Learning - bagging, boosting, stacking, Unsupervised learning: K-means, Instance Based Learning: KNN, Gaussian mixture models and Expectation maximization.

3.1 Combining multiple learners

In any application, we can use one of several learning algorithms, and with certain algorithms, there are hyperparameters that affect the final learner. Learning is an ill-posed problem and with finite data, each algorithm converges to a different solution and fails under different circumstances. The performance of a learner may be fine-tuned to get the highest possible accuracy on a validation set, but this finetuning is a complex task and still there are instances on which even the best learner is not accurate enough. The idea is that there may be another base-learner learner that is accurate on these. By suitably combining multiple ***baselearners***
then, accuracy can be improved.

There are basically two questions here:

1. How do we generate base-learners that complement each other?
2. How do we combine the outputs of base-learners for maximum accuracy?

The model combination is not a trick that always increases accuracy; model combination does always increase time and space complexity of training and testing, and unless base-learners are trained carefully and their decisions combined smartly, we will only pay for this extra complexity without any significant gain in accuracy. model combination is not a trick that always increases accuracy; model combination does always increase time and space complexity of training and testing, and unless base-learners are trained carefully and their decisions combined smartly, we will only pay for this extra complexity without any significant gain in accuracy.

Different Algorithms

We can use different learning algorithms to train different base-learners. Different algorithms make different assumptions about the data and lead to different classifiers.

Different Hyperparameters

We can use the same learning algorithm but use it with different hyperparameters. Examples are the number of hidden units in a multilayer perceptron, k in k-nearest neighbor, error threshold in decision trees, the kernel function in support vector machines, and so forth.

Different Input Representations

Separate base-learners may be using different *representations* of the same input object or event, making it possible to integrate different types of sensors/measurements/modalities. Different representations make different characteristics explicit allowing better identification. In many applications, there are multiple sources of information, and it is desirable to use all of these data to extract more information and achieve higher accuracy in prediction. For example, in speech recognition, to recognize the uttered words, in addition to the acoustic input, we can also use the video image of the speaker's lips and shape of the mouth as the words are spoken. This is similar to *sensor fusion* where the data from different sensors are integrated to extract more information for a specific application. Another example is information, for example, image retrieval where in addition to the image itself, we may also have text annotation in the form of keywords. In such a case, we want to be able to combine both of these sources to find the right set of images; this is also sometimes called ***multi-view learning***.

Different Training Sets

Another possibility is to train different base-learners by different subsets of the training set. This can be done randomly by drawing random training sets from the given sample; this is called *bagging*. Or, the learners can be trained serially so that instances on which the preceding base-learners are not accurate are given more emphasis in training later base-learners; examples are *boosting* and *cascading*, which actively try to generate complementary learners, instead of leaving this to chance.

Diversity vs. Accuracy

One important note is that when we generate multiple base-learners, we want them to be reasonably accurate but do not require them to be very accurate individually, so they are not, and need not be, optimized separately for best accuracy. The base-learners are not chosen for their accuracy, but for their simplicity. We do require, however, that the baselearners be diverse, that is, accurate on different instances, specializing in subdomains of the problem. What we care for is the final accuracy when the base-learners are combined, rather than the accuracies of the base-learners we started from.

3.2 Model Combination Schemes

There are also different ways the multiple base-learners are combined to generate the final output:

1) *Multiexpert combination* methods have base-learners that work in *parallel*. These methods can in turn be divided into two:
   _ In the *global* approach, also called *learner fusion*, given an input, all base-learners generate an output and all these outputs are used.
   Examples are *voting* and *stacking*.
   _ In the *local* approach, or *learner selection*, for example, in *mixture of experts*, there is a *gating* model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.
2) *Multistage combination* methods use a *serial* approach where the next combination base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident. An example is *cascading*.

Let us say that we have L base-learners. We denote by $dj(x)$ the prediction of base-learner Mj given the arbitrary dimensional input x. In the case of multiple representations, each Mj uses a different input representation xj . The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L|\Phi)$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.

when there are K outputs, for each learner there are $dji(x)$, $i = 1, \ldots, K$, $j = 1, \ldots, L$, and, combining them, we also generate K values, yi, $i = 1, \ldots, K$ and then for example in classification, we choose the class with the maximum yi value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

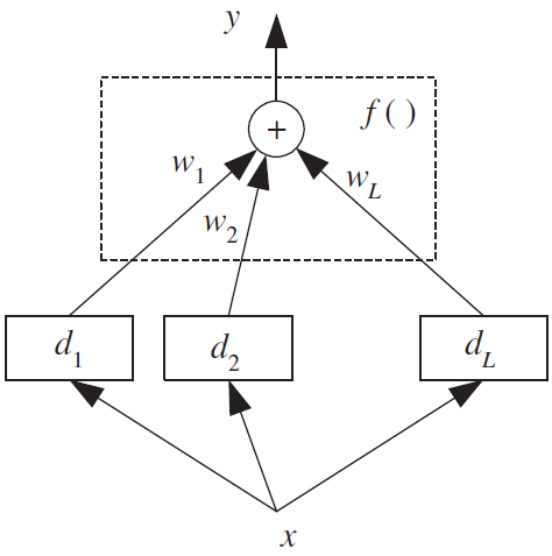


Figure 17.1 Base-learners are d_j and their outputs are combined using $f(\cdot)$. This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

3.3 Voting

The simplest way to combine multiple classifiers is by *voting*, which corresponds to taking a linear combination of the learners (see figure 17.1):

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1$$

This is also known as *ensembles* and *linear opinion pools*. In the simplest case, all learners are given equal weight and we have *simple voting*

Table 17.1 Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

Table 17.2 Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 17.1. If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale. An example of the use of these rules is shown in table 17.2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively.

With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0. Note that after the combination rules, y_i do not necessarily sum up to 1. In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $w_j = 1/L$.

In classification, this is called *plurality voting* where the class having the maximum number of votes is the winner. When there are two classes, this is *majority voting* where the winning class gets more than half of the votes (exercise 1). If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a *weighted voting* scheme. Equivalently, if d_{ji} are the class posterior probabilities, $P(C_i|x, \mathcal{M}_j)$, then we can just sum them up ($w_j = 1/L$) and choose the class with maximum y_i . In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average. Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods. This is *Bayesian model combination*. For example, in classification we have $w_j \equiv P(\mathcal{M}_j)$, $d_{ji} = P(C_i|x, \mathcal{M}_j)$, and equation 17.2 corresponds to

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j)$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe *P(*M*j)* is high, or we can have another Bayesian step and calculate *P(*M*j*|X*)*, the probability of a model given the sample, and sample high probable models from this density.

3.4 ENSEMBLE LEARNING

Ensemble learning in machine learning combines multiple individual models to create a stronger, more accurate predictive model. By leveraging the diverse strengths of different models, ensemble learning aims to mitigate errors, enhance performance, and increase the overall robustness of predictions, leading to improved results across various tasks in machine learning and data analysis.

Bagging (Bootstrap Aggregating) is an ensemble learning technique designed to improve the accuracy and stability of machine learning algorithms. It involves the following steps:

1. **Data Sampling**: Creating multiple subsets of the training dataset using bootstrap sampling (random sampling with replacement).
2. **Model Training**: raining a separate model on each subset of the data.
3. **Aggregation**: Combining the predictions from all individual models (averaged for regression or majority voting for classification) to produce the final output.

Key Benefits:

- **Reduces Variance**: By averaging multiple predictions, bagging reduces the variance of the model and helps prevent overfitting.
- **Improves Accuracy**: Combining multiple models usually leads to better performance than individual models.

Example of Bagging Algorithms:

- Random Forests (an extension of bagging applied to decision trees)

Boosting is another ensemble learning technique that focuses on creating a strong model by combining several weak models. It involves the following steps:

1. **Sequential Training**: Training models sequentially, each one trying to correct the errors made by the previous models.
2. **Weight Adjustment**: Each instance in the training set is weighted. Initially, all instances have equal weights. After each model is trained, the weights of misclassified instances are increased so that the next model focuses more on difficult cases.

3. **Model Combination**: Combining the predictions from all models to produce the final output, typically by weighted voting or weighted averaging.

Key Benefits:

- **Reduces Bias**: By focusing on hard-to-classify instances, boosting reduces bias and improves the overall model accuracy.
- **Produces Strong Predictors**: Combining weak learners leads to a strong predictive model.

Example of Boosting Algorithms:

- AdaBoost
- Gradient Boosting Machines (GBM)
- XGBoost
- LightGBM

Similarities between Bagging and Boosting

Bagging (Bootstrap Aggregating) and Boosting are both ensemble learning techniques designed to improve the performance of machine learning models by combining the predictions of multiple base models.

Both bagging and boosting in machine learning involve training multiple models on different subsets of the training data and then combining their predictions to make a final prediction. These techniques aim to reduce the variance of the model and improve its overall accuracy and stability.

Additionally, using bagging and boosting in machine learning with various base models, such as decision trees, to create a diverse set of models that capture different aspects of the data.

Differences between Bagging and Boosting

While bagging and boosting share some similarities, their approach and methodology differ.

Bagging trains each base model independently and in parallel, using bootstrap sampling to create multiple subsets of the training data. The final prediction is then made by averaging the predictions of all base models. Bagging focuses on reducing variance and overfitting by creating diverse models.

In contrast, boosting trains models sequentially, with each subsequent model focusing on correcting the errors made by the previous ones. Boosting adjusts the weights of training instances to prioritize difficult-to-classify instances, thus reducing bias and improving predictive accuracy. The final prediction is made by combining the predictions of all models, typically using a weighted voting or averaging approach.

Additionally, while bagging is relatively simple and easy to parallelize, boosting is more complex due to its sequential nature and may be more prone to overfitting if not properly controlled.

Stacking (Stacked Generalization) is an ensemble learning technique that aims to combine multiple models to improve predictive performance. It involves the following steps:

1. **Base Models**: Training multiple models (level-0 models) on the same dataset.
2. **Meta-Model**: Training a new model (level-1 or meta-model) to combine the predictions of the base models. Using the predictions of the base models as input features for the meta-model.

Key Benefits:

- **Leverages Model Diversity**: By combining different types of models, stacking can capture a wide range of patterns in the data.
- **Improves Performance**: The meta-model learns how to best combine the predictions from the base models, often leading to improved performance over individual models.

Example Process:

- Train several base models (e.g., decision trees, neural networks, SVMs) on the training data.
- Use the predictions of these base models to create a new dataset.
- Train a meta-model (e.g., linear regression, logistic regression) on this new dataset to make the final predictions.

How Did Ensemble Learning Come into Existence?

One of the first uses of ensemble methods in machine learning was the bagging technique. This technique was developed to overcome instability in decision trees. An example of the bagging technique is the random forest algorithm. The random forest is an ensemble of multiple decision trees. Decision trees tend to be prone to overfitting. Because of this, a single decision tree can't be relied on to make predictions. To improve the prediction accuracy of decision trees, bagging is employed to form a random forest. The resulting random forest has a lower variance compared to the individual trees.

The success of bagging led to developing other ensemble techniques such as boosting, stacking, and many others. Today, these developments are an important part of machine learning.

The many real-life machine learning applications show these ensemble methods in machine learning' importance. These applications include many critical systems. These include decision-making systems, spam detection, autonomous vehicles, medical diagnosis, etc. These systems are crucial because they can impact human lives and business revenues. Therefore, ensuring the accuracy of machine learning models is

paramount. An inaccurate model can lead to disastrous consequences for many businesses or organizations. At worst, they can lead to the endangerment of human lives.

How Ensemble Learning Works?

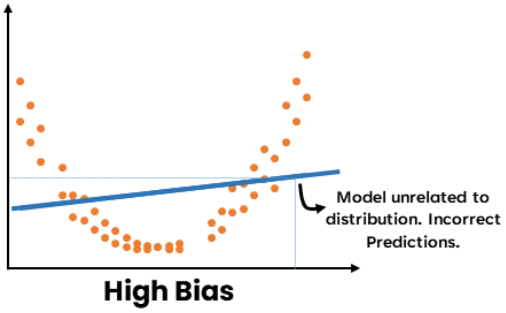
Ensemble learning is a learning method that consists of combining multiple machine learning models.

A problem in machine learning is that individual models tend to perform poorly. In other words, they tend to have low prediction accuracy. To mitigate this problem, we combine multiple models to get one with a better performance.

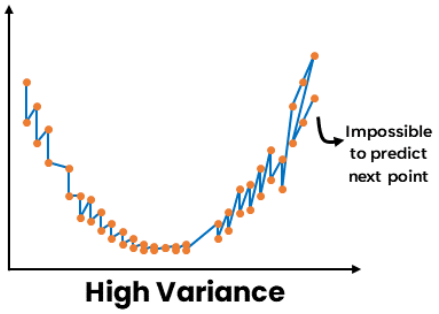
The individual models that we combine are known as weak learners. We call them weak learners because they either have a high bias or high variance. Because they either have high bias or variance, weak learners cannot learn efficiently and perform poorly.

High-bias and High-variance Models

- A high-bias model results from not learning data well enough. It is not related to the distribution of the data. Hence, future predictions will be unrelated to the data and thus incorrect.



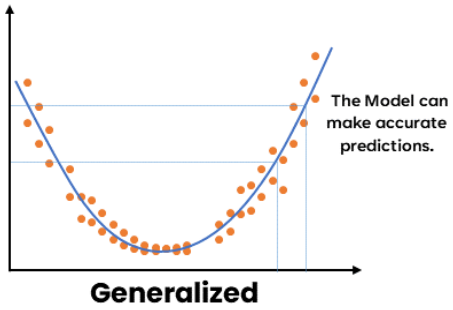
- A high-variance model results from learning the data too well. It varies with each data point, making it impossible to predict the next point accurately.



Thus, both high bias and high variance models cannot be generalized properly. Thus, weak learners will make incorrect generalizations or fail to generalize altogether. Because of this, the predictions of weak learners cannot be relied on by themselves.

As we know from the bias-variance trade-off, an underfit model has high bias and low variance, whereas an overfit model has high variance and low bias. In either case, there is no balance between bias and variance. For there to be a balance, both the bias and variance need to be low. Ensemble learning tries to balance this bias-variance trade-off by reducing either the bias or the variance.

Ensemble learning aims to reduce the bias if we have a weak model with high bias and low variance. This way, the resulting model will be much more balanced, with low bias and variance. Thus, the resulting model will be known as a strong learner. This model will be more generalized than the weak learners. It will thus be able to make accurate predictions.



Monitoring Ensemble Learning Models

We use bagging to combine weak learners of high variance. Bagging aims to produce a model with lower variance than the individual weak models. These weak learners are homogenous, meaning they are of the same type.

Bagging is also known as Bootstrap aggregating. It consists of two steps: bootstrapping and aggregation.

Bootstrapping

Involves resampling subsets of data with replacement from an initial dataset. In other words, the initial dataset provides subsets of data. Creating these subsets, bootstrapped datasets or simply bootstraps, by resampling 'with replacement,' which means an individual data point can be sampled multiple times. Each bootstrap dataset trains a weak learner.

Aggregating

Individual weak learners train independently from each other. Each learner makes independent predictions. The system aggregates the results of those predictions to get the overall prediction. The predictions are aggregated using either max voting or averaging.

Max Voting

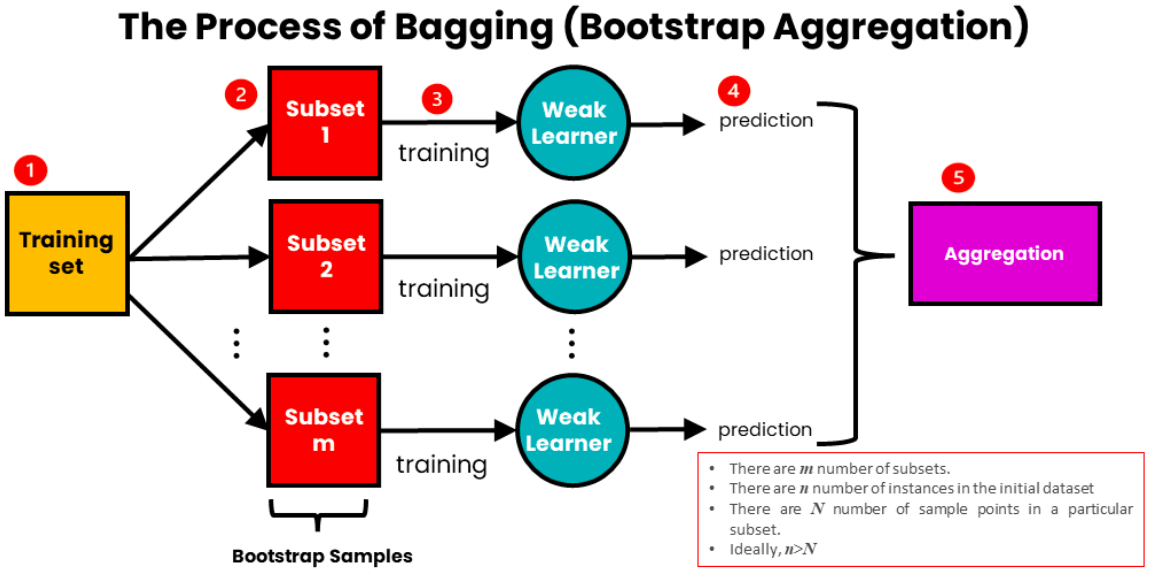
It is commonly used for classification problems to take the mode of the predictions (the most occurring prediction). It's called voting because, like in election voting, the premise

is that 'the majority rules.' Each model makes a prediction, and a prediction from each model counts as a single 'vote.' The most occurring 'vote' is chosen as the representative for the combined model.

Averaging

Using it generally for regression problems. It involves taking the average of the predictions. The resulting average is used as the overall prediction for the combined model.

Steps of Bagging



The steps of bagging are as follows:

1. We have an initial training dataset containing n-number of instances.
2. We create a m-number of subsets of data from the training set. We take a subset of N sample points from the initial dataset for each subset. Each subset is taken with replacement. This means that a specific data point can be sampled more than once.
3. For each subset of data, we train the corresponding weak learners independently. These models are homogeneous, meaning that they are of the same type.
4. Each model makes a prediction.
5. Aggregating the predictions into a single prediction. For this, using either max voting or averaging.

Reducing Bias by Boosting

We use boosting to combine weak learners with high bias. Boosting aims to produce a model with a lower bias than the individual models. Like in bagging, the weak learners are homogeneous.

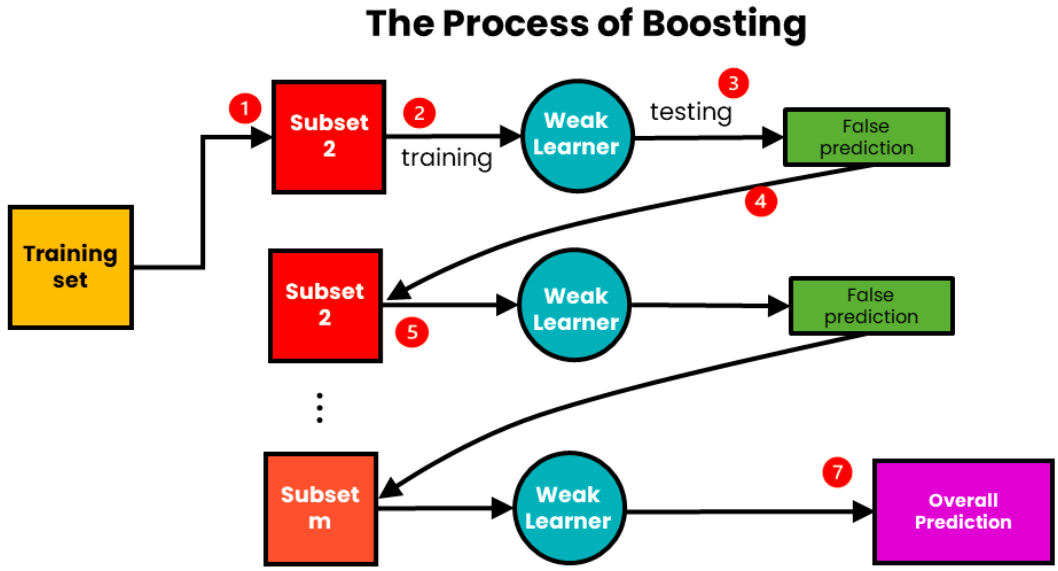
Boosting involves sequentially training weak learners. Here, each subsequent learner improves the errors of previous learners in the sequence. A sample of data is first taken

from the initial dataset. Using this sample to train the first model, and the model makes its prediction. The samples can either be correctly or incorrectly predicted. The samples that are wrongly predicted are reused for training the next model. In this way, subsequent models can improve on the errors of previous models.

Unlike bagging, which aggregates prediction results at the end, boosting aggregates the results at each step. They are aggregated using weighted averaging.

Weighted averaging involves giving all models different weights depending on their predictive power. In other words, it gives more weight to the model with the highest predictive power. This is because the learner with the highest predictive power is considered the most important.

Steps of Boosting



Boosting works with the following steps:

- We sample m-number of subsets from an initial training dataset.
- Using the first subset, we train the first weak learner.
- We test the trained weak learner using the training data. As a result of the testing, some data points will be incorrectly predicted.
- Each data point with the wrong prediction is sent into the second subset of data, and this subset is updated.
- Using this updated subset, we train and test the second weak learner.
- We continue with the following subset until the total number of subsets is reached.
- We now have the total prediction. The overall prediction has already been aggregated at each step, so there is no need to calculate it.

Improving Model Accuracy with Stacking

We use stacking to improve the prediction accuracy of strong learners. Stacking aims to create a single robust model from multiple heterogeneous strong learners.

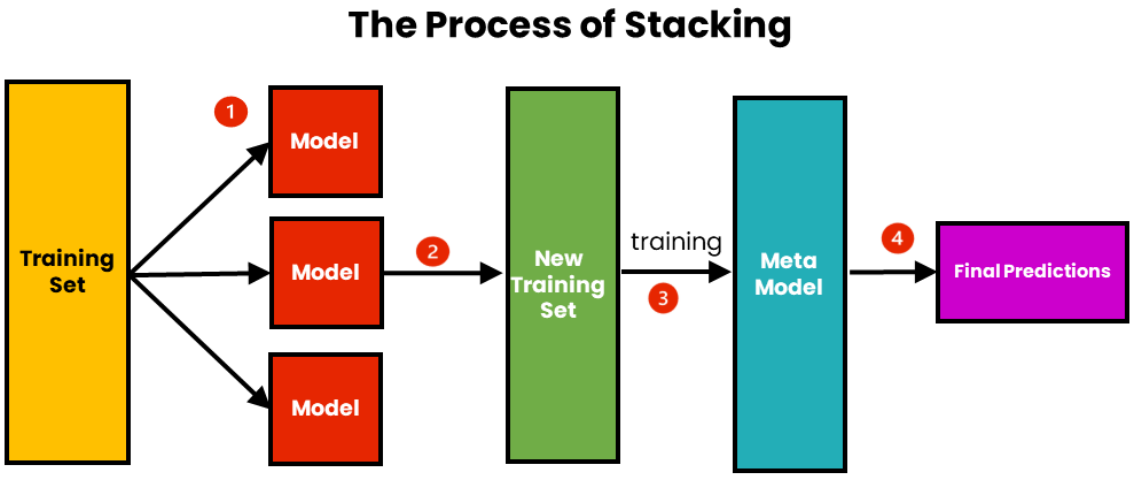
Stacking differs from bagging and boosting in machine learning in that:

- It combines strong learners
- It combines heterogeneous models
- It consists of creating a Metamodel.

Individual heterogeneous models are trained using an initial dataset. These models make predictions and form a single new dataset using those predictions. Using this new data set to train the metamodel, which makes the final prediction. Combining the predi ction using weighted averaging.

Because stacking combines strong learners, it can combine bagged or boosted models.

Steps of Stacking



The steps of Stacking are as follows:

- We use initial training data to train m-number of algorithms.
- Using the output of each algorithm, we create a new training set.
- Using the new training set, we create a meta-model algorithm.
- Using the results of the meta-model, we make the final prediction. Combining the result using weighted averaging.

3.5 UNSUPERVISED LEARNING - K-Means Clustering

K-Means Clustering is an Unsupervised Machine Learning algorithm, which groups the unlabeled dataset into different clusters. Unsupervised Machine Learning is the process of teaching a computer to use unlabeled, unclassified data and enabling the algorithm to operate on that data without supervision. Without any previous data training, the machine's job in this case is to organize unsorted data according to parallels, patterns, and variations.

K means clustering, assigns data points to one of the K clusters depending on their distance from the center of the clusters. It starts by randomly assigning the clusters centroid in the space. Then each data point assign to one of the cluster based on its distance from centroid of the cluster. After assigning each point to one of the cluster, new cluster centroids are assigned. This process runs iteratively until it finds good cluster. In the analysis we assume that number of cluster is given in advanced and we have to put points in one of the group.

In some cases, K is not clearly defined, and we have to think about the optimal number of K. K Means clustering performs best data is well separated. When data points overlapped this clustering is not suitable. K Means is faster as compare to other clustering technique. It provides strong coupling between the data points. K Means cluster do not provide clear information regarding the quality of clusters. Different initial assignment of cluster centroid may lead to different clusters. Also, K Means algorithm is sensitive to noise. It may have stuck in local minima.

Objective

The goal of clustering is to divide the population or set of data points into a number of groups so that the data points within each group are more comparable to one another and different from the data points within the other groups. It is essentially a grouping of things based on how similar and different they are to one another.

Working

We are given a data set of items, with certain features, and values for these features (like a vector). The task is to categorize those items into groups. To achieve this, we will use the K-means algorithm, an unsupervised learning algorithm. 'K' in the name of the algorithm represents the number of groups/clusters we want to classify our items into. (It will help if you think of items as points in an n-dimensional space). The algorithm will categorize the items into k groups or clusters of similarity. To calculate that similarity, we will use the Euclidean distance as a measurement.

The algorithm works as follows:

1. First, we randomly initialize k points, called means or cluster centroids.
2. We categorize each item to its closest mean, and we update the mean's coordinates, which are the averages of the items categorized in that cluster so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

The "points" mentioned above are called means because they are the mean values of the items categorized in them. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the

means at random values between the boundaries of the data set (if for a feature *x*, the items have values in [0,3], we will initialize the means with values for *x* at [0,3]).
The above algorithm in pseudocode is as follows:

```
Initialize k means with random values
--> For a given number of iterations:

   --> Iterate through items:

      --> Find the mean closest to the item by calculating
      the euclidean distance of the item with each of the means

      --> Assign item to mean

      --> Update mean by shifting it to the average of the items in that cluster
```

Implementation of K-Means Clustering in Python

Example 1

Import the necessary Libraries

We are importing Numpy for statistical computations, Matplotlib to plot the graph, and make_blobs from sklearn.datasets.

- Python3

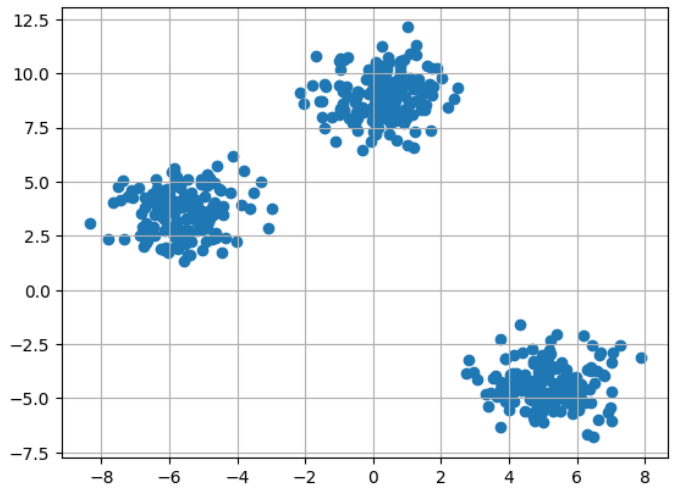
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
```

Create the custom dataset with make_blobs and plot it

- Python3

```
X,y = make_blobs(n_samples = 500,n_features = 2,centers = 3,random_state = 23)


fig = plt.figure(0)

plt.grid(True)

plt.scatter(X[:,0],X[:,1])

plt.show()
```

Output:



Clustering dataset

Initialize the random centroids

The code initializes three clusters for K-means clustering. It sets a random seed and generates random cluster centers within a specified range, and creates an empty list of points for each cluster.

- Python3

```
k = 3


clusters = {}
np.random.seed(23)


for idx in range(k):
    center = 2*(2*np.random.random((X.shape[1],))-1)
    points = []
    cluster = {
        'center' : center,
        'points' : []
    }


    clusters[idx] = cluster
```

```
clusters
```

Output:

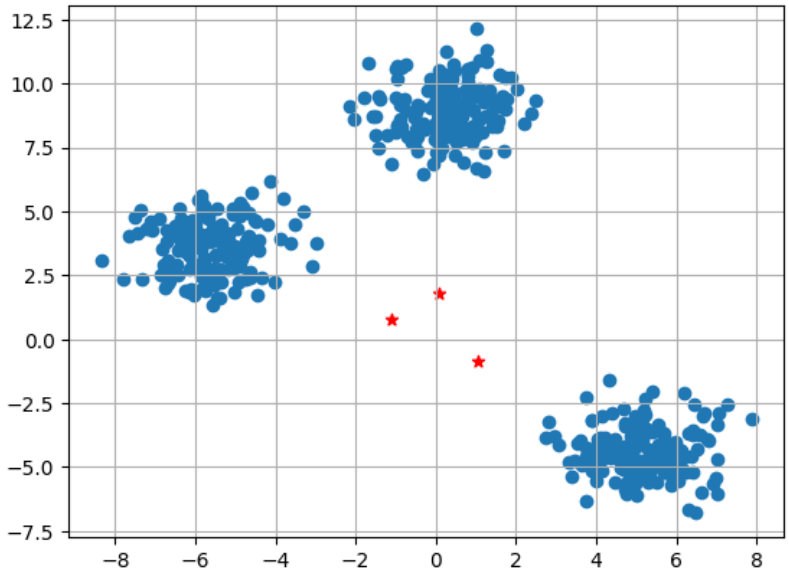
```
{0: {'center': array([0.06919154, 1.78785042]), 'points': []},
 1: {'center': array([ 1.06183904, -0.87041662]), 'points': []},
 2: {'center': array([-1.11581855,  0.74488834]), 'points': []}}
```

Plot the random initialize center with data points

- Python3

```
plt.scatter(X[:,0],X[:,1])
plt.grid(True)
for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '*',c = 'red')
plt.show()
```

Output:



Data points with random center

The plot displays a scatter plot of data points (X[:,0], X[:,1]) with grid lines. It also marks the initial cluster centers (red stars) generated for K-means clustering.

Define Euclidean distance

- Python3

```
def distance(p1,p2):
    return np.sqrt(np.sum((p1-p2)**2))
```

Create the function to Assign and Update the cluster center

The E-step assigns data points to the nearest cluster center, and the M-step updates cluster centers based on the mean of assigned points in K-means clustering.

- Python3

```
#Implementing E step
def assign_clusters(X, clusters):
    for idx in range(X.shape[0]):
        dist = []

        curr_x = X[idx]

        for i in range(k):
            dis = distance(curr_x,clusters[i]['center'])
            dist.append(dis)
        curr_cluster = np.argmin(dist)
        clusters[curr_cluster]['points'].append(curr_x)
    return clusters


#Implementing the M-Step
def update_clusters(X, clusters):
    for i in range(k):
        points = np.array(clusters[i]['points'])
        if points.shape[0] > 0:
            new_center = points.mean(axis =0)
            clusters[i]['center'] = new_center

            clusters[i]['points'] = []
```

```
    return clusters
```

Step 7: Create the function to Predict the cluster for the datapoints

- Python3

```
def pred_cluster(X, clusters):
    pred = []
    for i in range(X.shape[0]):
        dist = []
        for j in range(k):
            dist.append(distance(X[i],clusters[j]['center']))
        pred.append(np.argmin(dist))
    return pred
```

Assign, Update, and predict the cluster center

- Python3

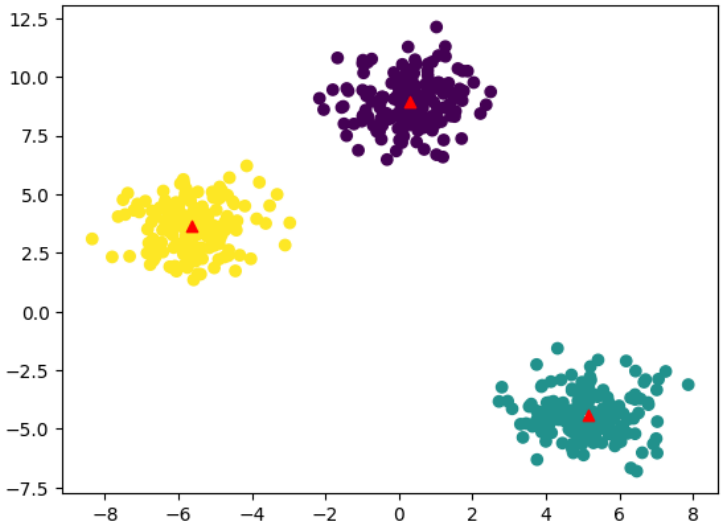
```
clusters = assign_clusters(X,clusters)
clusters = update_clusters(X,clusters)
pred = pred_cluster(X,clusters)
```

Plot the data points with their predicted cluster center

- Python3

```
plt.scatter(X[:,0],X[:,1],c = pred)
for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '^',c = 'red')
plt.show()
```

Output:



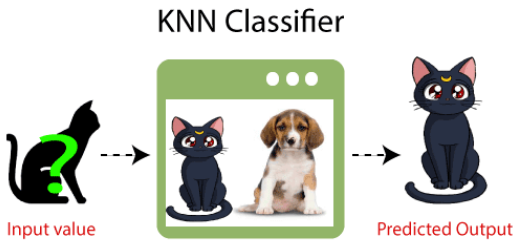
K-means Clustering

The plot shows data points colored by their predicted clusters. The red markers represent the updated cluster centers after the E-M steps in the K-means clustering algorithm.

3.6 INSTANCE BASED LEARNING: KNN ALGORITHM

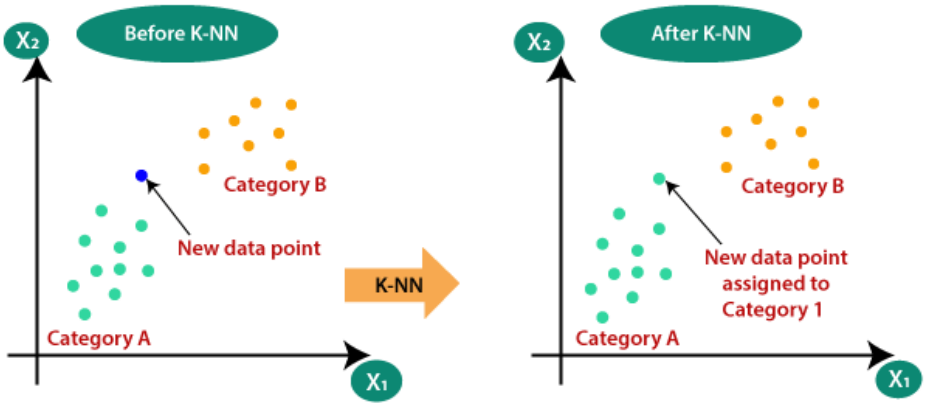
- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the

KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.



Why do we need a K-NN Algorithm?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:



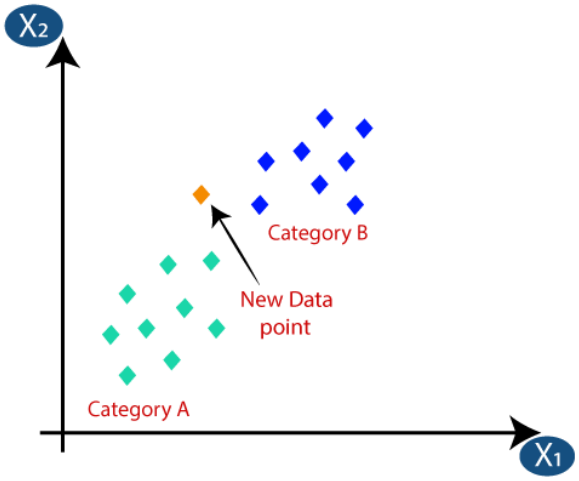
How does K-NN work?

The K-NN working can be explained on the basis of the below algorithm:

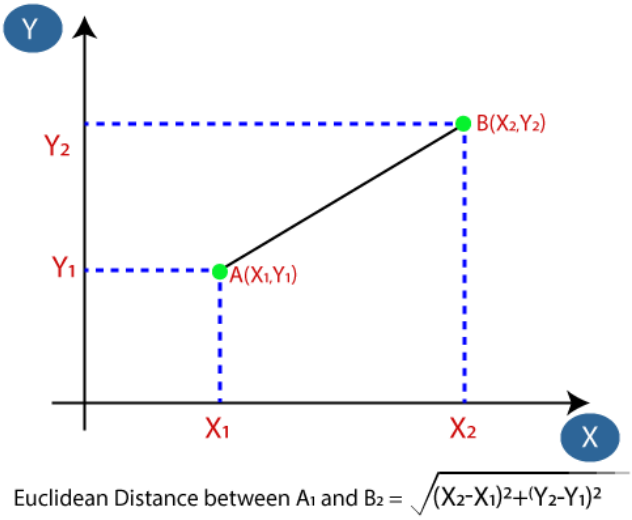
- **Step-1:** Select the number K of the neighbours
- **Step-2:** Calculate the Euclidean distance of **K number of neighbours**
- **Step-3:** Take the K nearest neighbours as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbours, count the number of the data points in each category.
- **Step-5:** Assign the new data points to that category for which the number of the neighbour is maximum.

- **Step-6:** Our model is ready.

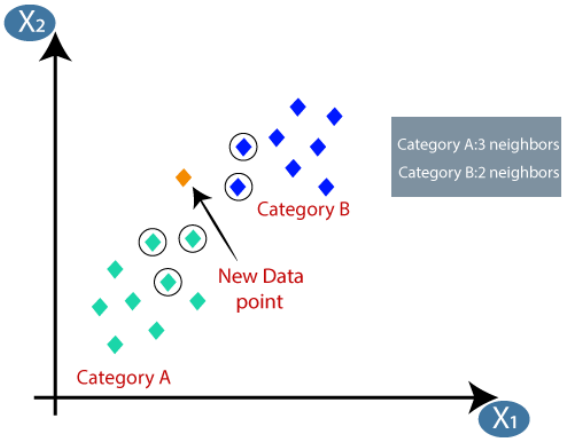
Suppose we have a new data point and we need to put it in the required category. Consider the below image:



- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

How to select the value of K in the K-NN Algorithm?

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

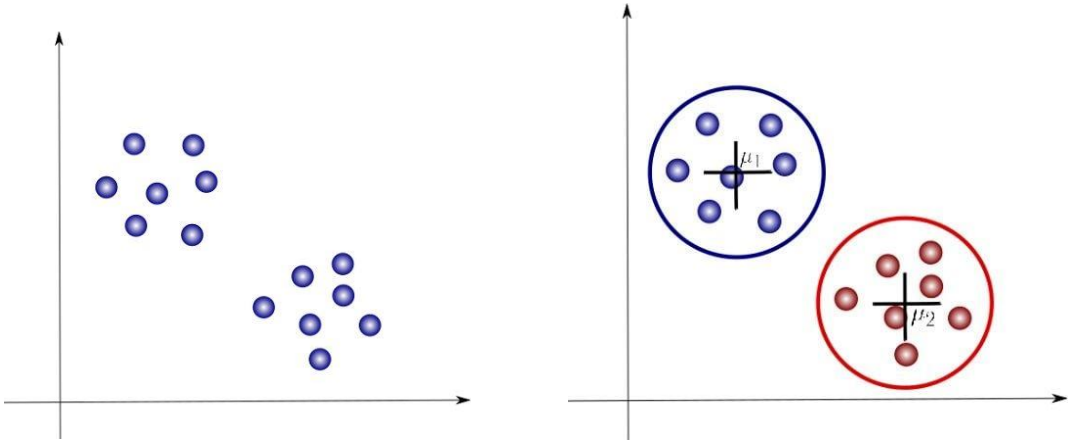
- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

3.7 GAUSSIAN MIXTURE MODEL

A Gaussian mixture model is a soft clustering technique used in unsupervised learning to determine the probability that a given data point belongs to a cluster. It's composed of several Gaussians, each identified by $k \in \{1,\ldots, K\}$, where K is the number of clusters in a data set.

A Gaussian mixture model is a machine learning method used to determine the probability each data point belongs to a given cluster. The model is a soft clustering method used in unsupervised learning.

Clustering is an unsupervised learning problem where we intend to find clusters of points in our data set that share some common characteristics. Let's suppose we have a data set that looks like this, Our job is to find sets of points that appear close together. In this case, we can clearly identify two clusters of points that we will color blue and red, respectively:



Two data clusters. | Image: Oscar Contreras Carrasco

Here, μ1 and μ2 are the centroids of each cluster and are parameters that identify each of these. Let's look at how a Gaussian mixture model can help us analyze these clusters.

We are now introducing some additional notation. Here, μ1 and μ2 are the centroids of each cluster and are parameters that identify each of these. Let's look at how a Gaussian mixture model can help us analyze these clusters.

Gaussian Mixture Model vs. K-Means

A popular clustering algorithm is known as K-means, which will follow an iterative approach to update the parameters of each cluster. More specifically, what it will do is compute the means (or centroids) of each cluster, and then calculate their distance to each of the data points. The latter are then labeled as part of the cluster that is identified by their closest centroid. This process is repeated until a convergence criterion is met, such as when we see no further changes in the cluster assignments.

One important characteristic of K-means is that it's a hard clustering method, which means that it will associate each point to one and only one cluster. A limitation to this approach is that

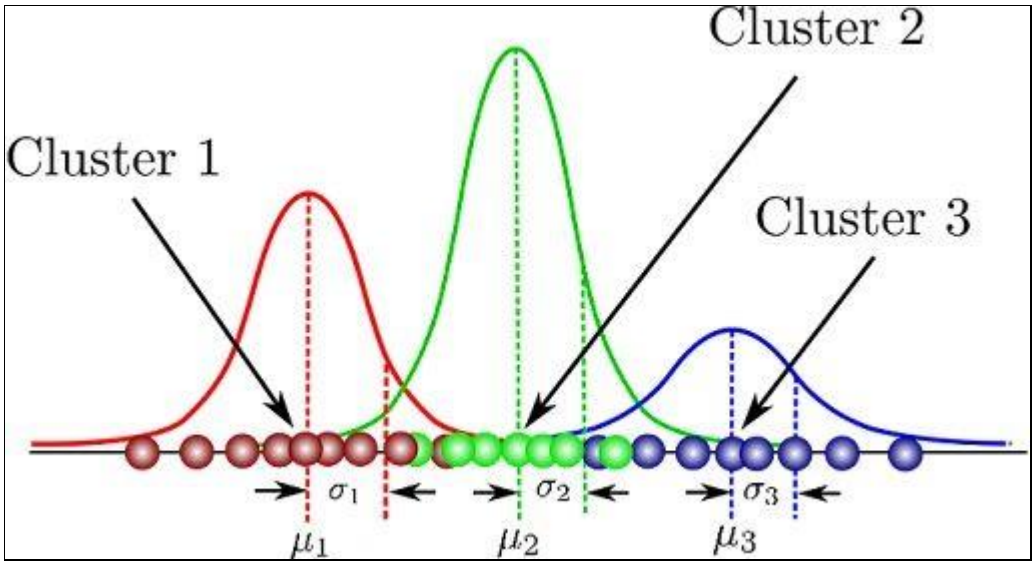
there is no uncertainty measure or probability that tells us how much a data point is associated with a specific cluster. So what about using a soft clustering instead of a hard one? This is exactly what Gaussian mixture models (GMMs) attempt to do. Let's now discuss this method further.

What Is a Gaussian Mixture Model?

A Gaussian mixture is a function that is composed of several Gaussians, each identified by $k \in \{1,\ldots, K\}$, where K is the number of clusters of our data set. Each Gaussian k in the mixture is comprised of the following parameters:

- A mean μ that defines its center.
- A covariance Σ that defines its width. This would be equivalent to the dimensions of an ellipsoid in a multivariate scenario.
- A mixing probability π that defines how big or small the Gaussian function will be.

Let's illustrate these parameters graphically:



Here, we can see that there are three Gaussian functions, hence $K = 3$. Each Gaussian explains the data contained in each of the three clusters available. The mixing coefficients are themselves probabilities and must meet this condition:

$$\sum_{k=1}^{K} \pi_k = 1 \qquad (1)$$

To determine the optimal values for these parameters we must ensure that each Gaussian fits the data points belonging to each cluster. This is exactly what maximum likelihood does.

In general, the Gaussian density function is given by:

$$\mathcal{N}(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

where x represents our data points, *D* is the number of dimensions of each data point. μ and Σ are the mean and covariance, respectively. If we have a data set composed of $N = 1000$ three-dimensional points ($D = 3$), then x will be a 1000×3 matrix. μ will be a 1×3 vector, and Σ will be a 3×3 matrix. For later purposes, we will also find it useful to take the log of this equation, which is given by:

$$\ln\mathcal{N}(\mathbf{x}|\mu,\Sigma) = -\frac{D}{2}\ln 2\pi - \frac{1}{2}\ln\Sigma - \frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu) \qquad (2)$$

If we differentiate this equation with respect to the mean and covariance and then equate it to zero, then we will be able to find the optimal values for these parameters, and the solutions will correspond to the maximum likelihood estimation (MLE) for this setting.

However, because we are dealing with not just one, but many Gaussians, it is complicated to find the parameters for the whole mixture.

First, let's suppose we want to know what is the probability that a data point x*n* comes from Gaussian *k*. We can express this as:

$$p(z_{nk} = 1|\mathbf{x}_n)$$

Data point xn equation expression. | Image: Oscar Contreras Carrasco

which reads "given a data point **x**, what is the probability it came from Gaussian k?" In this case, *z* is a latent variable that takes only two possible values. It is one when **x** comes from Gaussian *k*, and zero otherwise. Actually, we don't get to see this *z* variable in reality, but knowing its probability of occurrence will be useful in helping us determine the Gaussian mixture parameters, as we will discuss later.

Likewise, we can state the following:

$$\pi_k = p(z_k = 1)$$

Gaussian mixture probability equation. | Image: Oscar Contreras Carrasco

which means that the overall probability of observing a point that comes from Gaussian *k* is actually equivalent to the mixing coefficient for that Gaussian. This makes sense, because the bigger the Gaussian is, the higher we would expect this probability to be. Now, let z be the set of all possible latent variables *z*, hence:

$$\mathbf{z} = \{z_1, ..., z_K\}$$

Latent variable z equation. | Image: Oscar Contreras Carrasco

We know that each *z* occurs independently of others and that they can only take the value of one when *k* is equal to the cluster the point comes from. Therefore:

$$p(\mathbf{z}) = p(z_1 = 1)^{z_1} p(z_2 = 1)^{z_2} ... p(z_K = 1)^{z_K} = \prod_{k=1}^{K} \pi_k^{z_k}$$

Cluster point equation. | Image: Oscar Contreras Carrasco

what about finding the probability of observing our data given that it came from Gaussian *k*? Turns out that it's actually the Gaussian function itself! Following the same logic we used to define *p*(z), we can state:

$$p(\mathbf{x}_n|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)^{z_k}$$

Gaussian function equation for z. | Image: Oscar Contreras Carrasco

Remember, our initial aim was to determine the probability of *z* given our observation x. The equations we just derived, along with the Bayes rule, will help us determine this probability. From the product rule of probabilities, we know that:

$$p(\mathbf{x}_n, \mathbf{z}) = p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z})$$

Product rule of probabilities equation for z. | Image: Oscar Contreras Carrasco

The operands on the right are what we have just found. Perhaps some of you may be anticipating that we are going to use the Bayes rule to get the probability we eventually need. First we need *p*(x*n*), not *p*(x*n*, z). To get rid of z, we do Marginalization! We just need to sum up the terms on z, hence:

$$p(\mathbf{x}_n) = \sum_{k=1}^{K} p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

Summed up terms on z equation. | Image: Oscar Contreras Carrasco

This is the equation that defines a Gaussian mixture, and you can clearly see that it depends on all parameters that we mentioned previously! To determine the optimal values for these we need to determine the maximum likelihood of the model. We can find the likelihood as the joint probability of all observations x*n*, defined by:

$$p(\mathbf{X}) = \prod_{n=1}^{N} p(\mathbf{x}_n) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)$$

Probability for all observations equation. | Image: Oscar Contreras Carrasco

Like we did for the original Gaussian density function, let's apply the log to each side of the equation:

$$\ln p(\mathbf{X}) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k) \qquad (3)$$

Log applied to each side of the equation. | Image: Oscar Contreras Carrasco

Now, all we have to do is differentiate this equation with respect to the parameters to find the optimal parameters for the Gaussian mixture. We have an issue here. We can see that there is a logarithm that is affecting the second summation. Calculating the derivative of this expression and then solving for the parameters is going to be very hard.

We need to use an iterative method to estimate the parameters. But first, remember we were supposed to find the probability of *z* given x? At this point we already have everything in place to define what this probability will look like.

From Bayes rule, we know that:

$$p(z_k = 1 | \mathbf{x}_n) = \frac{p(\mathbf{x}_n | z_k = 1) p(z_k = 1)}{\sum_{j=1}^{K} p(\mathbf{x}_n | z_j = 1) p(z_j = 1)}$$

Bayes rule equation. | Image: Oscar Contreras Carrasco

From our earlier derivations we learned that:

$$p(z_k = 1) = \pi_k, \qquad p(\mathbf{x}_n | z_k = 1) = \mathcal{N}(\mathbf{x}_n | \mu_k, \mathbf{\Sigma}_k)$$

Previous derivation equations. | Image: Oscar Contreras Carrasco

So, let's now replace these in the previous equation:

$$p(z_k = 1 | \mathbf{x}_n) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n | \mu_j, \Sigma_j)} = \gamma(z_{nk}) \qquad (4)$$

Equation four with derivations placed into it. | Image: Oscar Contreras Carrasco

Gaussian Mixture Model Expectation-Maximization (EM) Algorithm

At this point, we've derived some expressions for the probabilities that we will find useful in determining the parameters of our model. However, in the past section, we could see that simply evaluating equation (3) to find such parameters would prove to be very hard. Fortunately, there is an iterative method we can use to achieve this purpose.

It's called the expectation — maximization (EM) algorithm. It's widely used for optimization problems where the objective function has complexities such as the one we've just encountered for the GMM case. Let the parameters of our model be:

$$\theta = \{\pi, \mu, \Sigma\}$$

Parameter equation for Gaussian mixture model. | Image: Oscar Contreras Carrasco

Let's now define the steps that the general EM algorithm will follow.

Step 1

Initialize θ accordingly. For instance, we can use the results obtained by a previous K-means run as a good starting point for our algorithm.

Step 2: Expectation Step

Evaluate:

$$\mathcal{Q}(\theta^*, \theta) = \mathbb{E}[\ln p(\mathbf{X}, \mathbf{Z}|\theta^*)] = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \theta) \ln p(\mathbf{X}, \mathbf{Z}|\theta^*) \qquad (5)$$

Expectation step equation. | Image: Oscar Contreras Carrasco

We've already found $p(Z|X, \theta)$. Remember the γ expression we ended up with in the previous section? For better visibility, let's bring our earlier equation (4) here:

$$p(z_k = 1|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \Sigma_j)} = \gamma(z_{nk}) \qquad (4)$$

Equation four expression. | Image: Oscar Contreras Carrasco

For Gaussian mixture models, the expectation step boils down to calculating the value of γ in equation (4) by using the old parameter values. Now if we replace equation (4) in equation (5), we will have:

$$\mathcal{Q}(\theta^*, \theta) = \sum_{\mathbf{Z}} \gamma(z_{nk}) \ln p(\mathbf{X}, \mathbf{Z}|\theta^*) \qquad (6)$$

Equation six created by combining equation four and five. | Image: Oscar Contreras Carrasco

But we are still missing $p(X, Z|\theta^*)$. How do we find it? Well, it's actually not that difficult. It's just the complete likelihood of the model, including both X and Z. We can find it by using the following expression:

$$p(\mathbf{X}, \mathbf{Z}|\theta^*) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi^{z_{nk}} \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)^{z_{nk}}$$

Likelihood for the model. | Image: Oscar Contreras Carrasco

Which is the result of calculating the joint probability of all observations and latent variables and is an extension of our initial derivations for $p(x)$. The log of this expression is given by:

$$\ln p(\mathbf{X}, \mathbf{Z}|\theta^*) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} [\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)] \qquad (7)$$

Log of the expression, creating equation 7. | Image: Oscar Contreras Carrasco

We've finally eliminated this troublesome logarithm that affected the summation in (3). With all of this in place, it will be much easier for us to estimate the parameters by just maximizing Q with respect to the parameters. We'll deal with that in the maximization step. Remember that the latent variable z will only be 1 once each time the summation is evaluated. With that knowledge, we can easily get rid of it as needed for our derivations.

Finally, we can replace equation (7) in (6) to get:

$$\mathcal{Q}(\theta^*, \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) [\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)] \qquad (8)$$

Replacing equation seven in six to create equation eight. | Image: Oscar Contreras Carrasco

In the maximization step, we will find the revised parameters of the mixture. For this purpose, we will need to make Q a restricted maximization problem and thus we will add a Lagrange multiplier to (8). Let's now review the maximization step.

Step 3: Maximization Step

Find the revised parameters θ^* using:

$$\theta^* = \arg\max_{\theta} \mathcal{Q}(\theta^*, \theta)$$

Revised parameters equation. | Image: Oscar Contreras Carrasco

Where:

$$\mathcal{Q}(\theta^*, \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) [\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)] \qquad (8)$$

Revised parameter equation eight. | Image: Oscar Contreras Carrasco

Which is what we ended up with in the previous step. However, Q should also take into account the restriction that all π values should sum up to one. To do so, we will need to add a suitable Lagrange multiplier. Therefore, we should rewrite equation (8) in this way:

$$\mathcal{Q}(\theta^*, \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) [\ln \pi_k + \ln \mathcal{N}(x_n | \mu_k, \Sigma_k)] - \lambda \left(\sum_{k=1}^{K} \pi_k - 1 \right) \qquad (8)$$

Lagrange multiplier added to the parameter equation. | Image: Oscar Contreras Carrasco

And now, we can easily determine the parameters by using maximum likelihood. Let's now take the derivative of Q with respect to π and set it equal to zero:

$$\frac{\partial \mathcal{Q}(\theta^*, \theta)}{\partial \pi_k} = \sum_{n=1}^{N} \frac{\gamma(z_{nk})}{\pi_k} - \lambda = 0$$

Derivative of Q equation set to 0. | Image: Oscar Contreras Carrasco

Then, by rearranging the terms and applying a summation over k to both sides of the equation, we obtain:

$$\sum_{n=1}^{N} \gamma(z_{nk}) = \pi_k \lambda \implies \sum_{k=1}^{K} \sum_{n=1}^{N} \gamma(z_{nk}) = \sum_{k=1}^{K} \pi_k \lambda$$

Solving for Q equation. | Image: Oscar Contreras Carrasco

From equation (1), we know that the summation of all mixing coefficients π equals one. In addition, we know that summing up the probabilities γ over k will also give us 1. Thus we get $\lambda = N$. Using this result, we can solve for π:

$$\pi_k = \frac{\sum_{n=1}^{N} \gamma(z_{nk})}{N}$$

Solving for pi equation. | Image: Oscar Contreras Carrasco

Similarly, if we differentiate Q with respect to μ and Σ, equate the derivative to zero and then solve for the parameters by making use of the log-likelihood equation (2) we defined, we obtain:

$$\mu_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n}{\sum_{n=1}^{N} \gamma(z_{nk})}, \qquad \Sigma_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) (\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma(z_{nk})}$$

Solving for parameters using log-likelihood equation. | Image: Oscar Conteras Carrasco

We'll use these revised values to determine γ in the next EM iteration and so on and so forth until we see some convergence in the likelihood value. We can use equation (3) to monitor the log-likelihood in each step and we are always guaranteed to reach a local maximum.