The **Apriori algorithm** repeatedly uses ***Apriori-gen* algorithm to generate candidates** and **then count their supports by reading the entire database once**.

```
Algorithm:  Apriori algorithm
Input:  a database and a user-defined minimum support
Output:  all frequent itemsets
 1. $L_0 := \emptyset$; $k := 1$;
 2. $C_1 := \{\{i\} \mid i \in I\}$
 3. Answer := $\emptyset$
 4. while $C_k \neq \emptyset$
 5.   read database and count supports for $C_k$
 6.   $L_k$ := {frequent itemsets in $C_k$}
 7.   $C_{k+1}$ := $Apriori\text{-}gen(L_k)$
 8.   $k := k + 1$
 9.   Answer := Answer $\cup$ $L_k$
10. return Answer
```

Apriori-gen relies on Property 1 mentioned above.

The candidate generation algorithm consists of a *join* procedure and a *prune* procedure

The *join* procedure combines two frequent *k*-itemsets, which have the same (k −1)-prefix, to generate a (k +1)-itemset as a new preliminary candidate.

Following the *join* procedure, the *prune* procedure is used to remove from the preliminary candidate set all itemsets *c* such that some *k*-subset of *c* is not a frequent itemset.

```
Algorithm:  The join procedure of the Apriori-gen algorithm
Input:  $L_k$, the set containing frequent itemsets found in pass $k$
Output:  preliminary candidate set $C_{k+1}$
/* The itemsets in $L_k$ are sorted */
1. for $i$ from 1 to $|L_k - 1|$
2.  for $j$ from $i + 1$ to $|L_k|$
3.   if $L_k.itemset_i$ and $L_k.itemset_j$ have the same $(k-1)$-prefix
4.     $C_{k+1}$ := $C_{k+1}$ $\cup$ $\{L_k.itemset_i \cup L_k.itemset_j\}$
5.   else
6.     break
```

```
Algorithm:  The prune procedure of the Apriori-gen algorithm
Input:  Preliminary candidate set $C_{k+1}$ generated from the join procedure above
Output:  final candidate set $C_{k+1}$ which does not contain any infrequent subset
1. for all itemsets $c$ in $C_{k+1}$
2.   for all $k$-subsets $s$ of $c$
3.     if $s \notin L_k$
4.       delete $c$ from $C_{k+1}$
```

The **bottom-up approach is good for the case** when ***all*** **maximal frequent itemsets are short**.

The **top-down approach is good** ***when all maximal frequent itemsets are long***.

If some maximal frequent itemsets are long and some are short, then **both one-way search approaches will not be efficient**.

The **bottom-up approach uses only Property 1** to reduce the number of candidates.

The **top-down approach uses only Property 2** to reduce the number of candidates

In our ***Pincer-Search*** **approach** we combine the top-down and the bottom-up searches.

We synergistically **rely on *both* properties to prune candidates**.

A **key component of the approach is the use of information gathered in the search in one direction to prune more candidates during the search in the other direction**.

If some maximal frequent itemset is found in the top-down direction, then this itemset can be used to eliminate (possibly many) candidates in the bottom-up direction.

The subsets of this frequent itemset can be pruned because they are frequent (Property 2).

Of course, if an infrequent itemset is found in the bottom-up direction, then it can be used to eliminate some candidates in the top-down direction (Property 1).

This "two-way search approach" can fully make use of both properties and thus speed up the search for the maximum frequent set.

Two-Way Search by Using the MFCS

In each pass:

- ❖ The algorithm counts the support of the candidates in the bottom-up direction
- ❖ The algorithm also counts supports of the itemsets in the MFCS: this set is adapted for the top-down search.

This will help in pruning candidates, but will also require changes in candidate generation.

Consider a pass *k*, during which, in the bottom-up direction, itemsets of size *k* are to be classified.

If, during the top-down direction some itemset that is an element of the MFCS of cardinality greater than *k* is found to be frequent, then all its subsets of cardinality *k* can be pruned from the set of candidates considered in the bottom-up direction in this pass.

They, and their supersets will never be candidates throughout the rest of the execution, potentially improving performance.

But of course, as the maximum frequent set is ultimately computed, they "will not be forgotten."

Similarly, when a new infrequent itemset is found in the bottom-up direction, the algorithm will use it to update the MFCS.

The subsets of the MFCS must not contain this infrequent itemset.

The MFCS is initialized to contain a single element, the itemset of cardinality *n* containing all the elements of the database.

As an example of its utility, consider the first pass of the bottom-up search.

If some *m* 1-itemsets are infrequent after the first pass (after reading the database once), the MFCS will have one element of cardinality $n-m$.

This itemset is generated by removing the *m* infrequent items from the initial element of the MFCS.

In this case, the top-down search goes down *m* levels in one pass.

In general, unlike the search in the bottom-up direction, which goes up one level in one pass, *the top-down search can go down many levels in one pass*.

Supervised Learning

Asks the machine to learn from our data when we specify a target variable.

This reduces the machine's task to only divining some pattern from the input data to get the target variable.

We address two cases of the target variable.
The first case occurs when the target variable can take only nominal values: true or false; reptile, fish, mammal, amphibian, plant, fungi.

The second case of classification occurs when the target variable can take an infinite number of numeric values, such as 0.100, 42.001, 1000.743, ... This case is called **Regression**.

Classifying with distance measurements - k-Nearest Neighbors

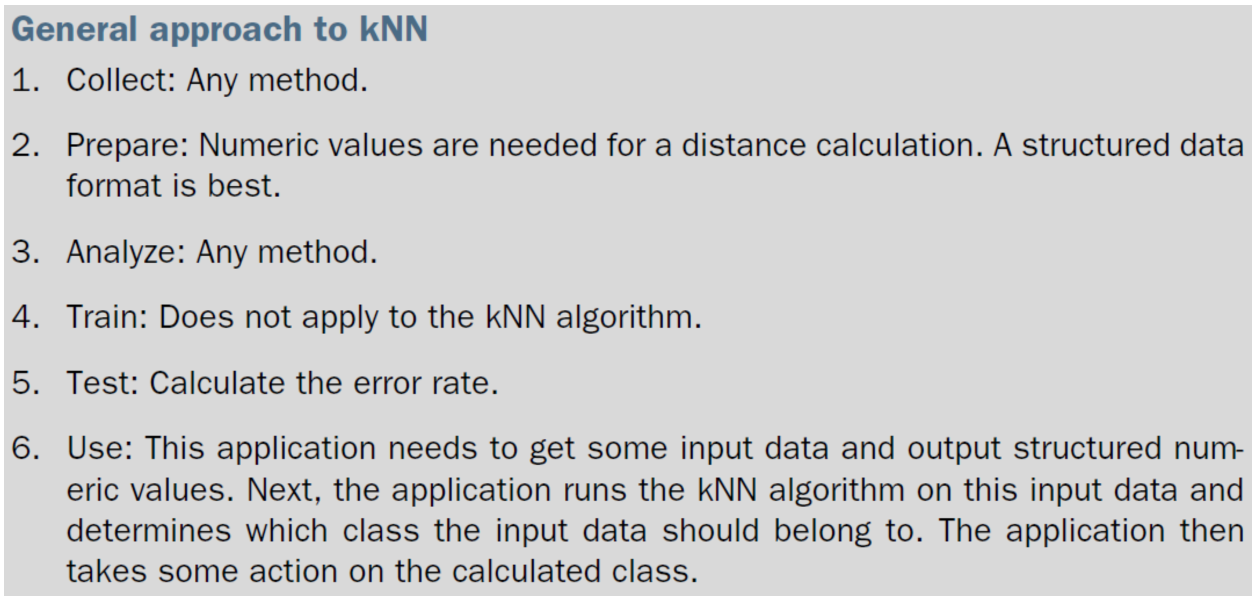
Pros: High accuracy, insensitive to outliers, no assumptions about data

Cons: Computationally expensive, requires a lot of memory

Works with: Numeric values, nominal values.

k-NN Working Principles:

- We have an existing set of example data, our training set.
- We have labels for all of this data - We know what class each piece of the data should fall into.
- When we're given a new piece of data without a label, we compare that new piece of data to the existing data, every piece of existing data.
- We then take the most similar pieces of data (the nearest neighbors) and look at their labels.
- We look at the top k most similar pieces of data from our known dataset; this is where the *k* comes from. (*k* is an integer and it's usually less than 20.)
- Lastly, we take a majority vote from the k most similar pieces of data, and the majority is the new class we assign to the data we were asked to classify.



General approach to kNN

1. Collect: Any method.
2. Prepare: Numeric values are needed for a distance calculation. A structured data format is best.
3. Analyze: Any method.
4. Train: Does not apply to the kNN algorithm.
5. Test: Calculate the error rate.
6. Use: This application needs to get some input data and output structured numeric values. Next, the application runs the kNN algorithm on this input data and determines which class the input data should belong to. The application then takes some action on the calculated class.

The data mining tasks can be broadly classified in two categories: descriptive and predictive.
Descriptive mining tasks characterize the general properties of the data in the database.
Predictive mining tasks perform inference on the current data in order to make predictions.

According to different goals, the mining task can be mainly divided into four types:

- ❖ Class/Concept Description,
- ❖ Association Analysis,
- ❖ Classification or Prediction and
- ❖ Clustering Analysis.

Feature Selection

Many irrelevant attributes may be present in data to be mined.
So they need to be removed.
Also many mining algorithms don’t perform well with large amounts of features or attributes. Therefore **Feature Selection Techniques** **needs to be applied before any kind of mining algorithm is applied**.
The main objectives of feature selection are to **avoid overfitting** and **improve model performance** and to provide faster and more cost-effective models.

Overfitting: In statistics and machine learning, one of the most common tasks is to fit a "model" to a set of training data, so as to be able to make reliable predictions on general untrained data.

In overfitting, **a statistical model describes random error or noise instead of the underlying relationship**.

Overfitting occurs when *a model is excessively complex*, such as **having too many parameters relative to the number of observations**.

A model that has been overfit ***has poor predictive performance***, **as it overreacts to minor fluctuations in the training data**.

The possibility of overfitting exists because the criterion used for training the model is not the same as the criterion used to judge the efficacy of a model.

In particular, a model is typically trained by maximizing its performance on some set of training data.

However, its efficacy is determined not by its performance on the training data but by its ability to perform well on unseen data.

Overfitting occurs when a model begins to "memorize" training data rather than "learning" to generalize from trend.

An extreme example: If the number of parameters is the same as or greater than the number of observations, a simple model or learning process can perfectly predict the training data simply by memorizing the training data in its entirety, but such a model will typically fail drastically when making predictions about new or unseen data, since the simple model has not learned to generalize at all.

The **potential for overfitting depends on**:

- Number of parameters and data
- The conformability of the model structure with the data shape, and
- The magnitude of model error compared to the expected level of noise or error in the data.

In order **to avoid overfitting**, it is necessary to use additional techniques like:

- Cross-Validation,
- Regularization,
- Early Stopping,
- Pruning,
- Bayesian priors on parameters or model comparison,

that can indicate when further training is not resulting in better generalization.

A **good analogy for the overfitting problem**:

- Imagine a baby trying to learn what is a window or what is not a window,
- We start to show him windows and he detects at an initial phase that all windows have glasses, and a frame and you can look outside, some of them may be opened.
- If we keep showing the same windows the baby may also falsely deduce that all windows are green, and that all green frames are windows.
- Thus overfitting the problem.