Big Data Analytics (BDA)

Big Data Mining Algorithms

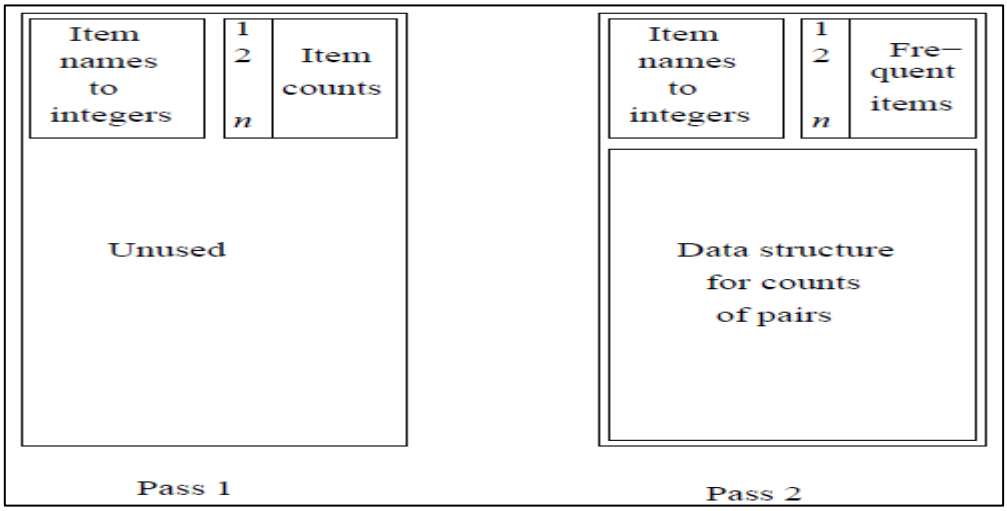
Frequent pattern mining

Handling larger datasets in main memory

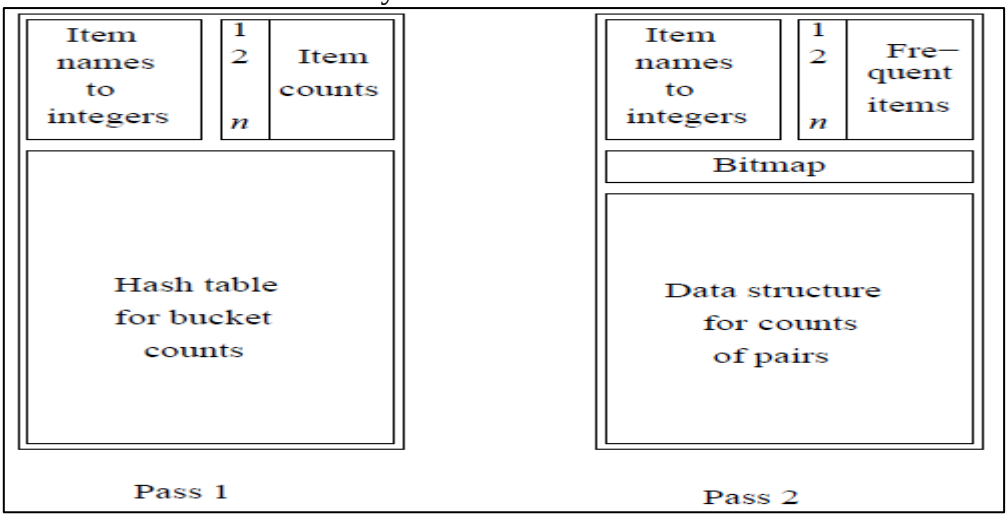
The A-Priori Algorithm is fine as long as the step with the greatest requirement for main memory – typically the counting of the candidate pairs C2 – has enough memory that it can be accomplished without thrashing (repeated moving of data between disk and main memory). Several algorithms have been proposed to cut down on the size of candidate set C2. Here, we consider the PCY Algorithm, which takes advantage of the fact that in the first pass of A-Priori there is typically lots of main memory not needed for the counting of single items. Then we look at the Multistage Algorithm, which uses the PCY trick and also inserts extra passes to further reduce the size of C2.

The Algorithm of Park, Chen, and Yu

This algorithm, which we call PCY after its authors, exploits the observation that there may be much unused space in main memory on the first pass. If there are a million items and gigabytes of main memory, we do not need more than 10% of the main memory for the two tables suggested as follows



A translation table from item names to small integers and an array to count those integers. The PCY Algorithm uses that space for an array of integers that generalizes the idea of a Bloom filter. The idea is shown schematically as follows



Think of this array as a hash table, whose buckets hold integers rather than sets of keys (as in an ordinary hash table) or bits (as in a Bloom filter). Pairs of items are hashed to buckets of this hash table. As we examine a basket during the first pass, we not only add 1 to the count for each item in the basket, but we generate all the pairs, using a double loop. We hash each pair, and we add 1 to the bucket into which that pair hashes. Note that the pair itself doesn't go into the bucket; the pair only affects the single integer in the bucket. At the end of the first pass, each bucket has a count, which is the sum of the counts of all the pairs that hash to that bucket. If the count of a bucket is at least as great as the support threshold s, it is called a frequent bucket. We can say nothing about the pairs that hash to a frequent bucket; they could all be frequent pairs from the information available to us. But if the count of the bucket is less than s (an infrequent bucket), we know no pair that hashes to this bucket can be frequent, even if the pair consists of two frequent items.
That fact gives us an advantage on the second pass. We can define the set of candidate pairs C2 to be those pairs {i, j} such that:
1. i and j are frequent items.
2. {i, j} hashes to a frequent bucket.
It is the second condition that distinguishes PCY from A-Priori.

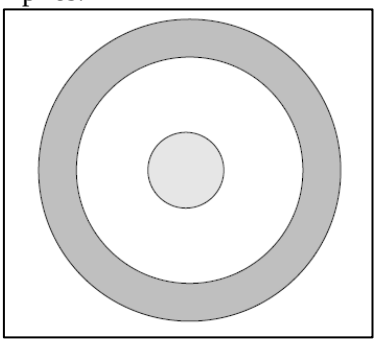
The Algorithm of Savasere, Omiecinski, and Navathe (SON)
It avoids both false negatives and false positives, at the cost of making two full passes. It is called the SON Algorithm after the authors. The idea is to divide the input file into chunks (which may be "chunks" in the sense of a distributed file system, or simply a piece of the file). Treat each chunk as a sample, and run the simple randomized algorithm on that chunk. We use *ps* as the threshold, if each chunk is fraction p of the whole file, and s is the support threshold. Store on disk all the frequent itemsets found for each chunk.
Once all the chunks have been processed in that way, take the union of all the itemsets that have been found frequent for one or more chunks. These are the candidate itemsets. Notice that if an itemset is not frequent in any chunk, then its support is less than *ps* in each chunk. Since the number of chunks is 1/p, we conclude that the total support for that itemset is less than *(1/p)ps = s*. Thus, every itemset that is frequent in the whole is frequent in at least one chunk, and we can be sure that all the truly frequent itemsets are among the candidates; i.e., there are no false negatives.
We have made a total of one pass through the data as we read each chunk and processed it. In a second pass, we count all the candidate itemsets and select those that have support at least s as the frequent itemsets.
The SON Algorithm and MapReduce
The SON algorithm lends itself well to a parallel-computing environment. Each of the chunks can be processed in parallel, and the frequent itemsets from each chunk combined to form the candidates. We can distribute the candidates to many processors, have each processor count the support for each candidate in a subset of the baskets, and finally sum those supports to get the support for each candidate itemset in the whole dataset. This process does not have to be implemented in MapReduce, but there is a natural way of expressing each of the two passes as a MapReduce operation.
First Map Function: Take the assigned subset of the baskets and find the itemsets frequent in the subset using the simple randomized algorithm. As described there, lower the support threshold from *s* to *ps* if each Map task gets fraction p of the total input file. The output is a set of key-value pairs (F, 1), where F is a frequent itemset from the sample. The value is always 1 and isirrelevant.

First Reduce Function: Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times. Thus, the output of the first Reduce function is the candidate itemsets.
Second Map Function: The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key-value pairs (C, v), where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.
Second Reduce Function: The Reduce tasks take the itemsets they are given as keys and sum the associated values. The result is the total support for each of the itemsets that the Reduce task was assigned to handle. Those itemsets whose sum of values is at least s are frequent in the whole dataset, so the Reduce task outputs these itemsets with their counts. Itemsets that do not have total support at least s are not transmitted to the output of the Reduce task.

Clustering Algorithms

CURE (Clustering Using REpresentative) Algorithm

It is a large-scale-clustering algorithm in the point-assignment class. This algorithm, called CURE (Clustering Using REpresentatives), assumes a Euclidean space. However, it does not assume anything about the shape of clusters; they need not be normally distributed, and can even have strange bends, S-shapes, or even rings. Instead of representing clusters by their centroid, it uses a collection of representative points, as the name implies.



Two Clusters, one surrounding the other

We begin the CURE algorithm by:
1. Take a small sample of the data and cluster it in main memory. In principle, any clustering method could be used, but as CURE is designed to handle oddly shaped clusters, it is often advisable to use a hierarchical method in which clusters are merged when they have a close pair of points.
2. Select a small set of points from each cluster to be representative points. These points should be chosen to be as far from one another as possible, using the method K-means.
3. Move each of the representative points a fixed fraction of the distance between its location and the centroid of its cluster. Perhaps 20% is a good fraction to choose. Note that this step requires a Euclidean space, since otherwise, there might not be any notion of a line between two points.
The next phase of CURE is to merge two clusters if they have a pair of representative points, one from each cluster, that are sufficiently close. The user may pick the distance that defines "close." This merging step can repeat, until there are no more sufficiently close clusters.

Canopy Clustering

The canopy clustering algorithm is an unsupervised pre-clustering algorithm introduced by Andrew McCallum, Kamal Nigam and Lyle Ungar in 2000. It is often used as preprocessing step for the K-means algorithm or the Hierarchical clustering algorithm. It is intended to speed up clustering operations on large data sets, where using another algorithm directly may be impractical due to the size of the data set.

The algorithm proceeds as follows, using two thresholds T_1 (the loose distance) and T_2 (the tight distance), where $T_1 > T_2$

1. Begin with the set of data points to be clustered.
2. Remove a point from the set, beginning a new 'canopy' containing this point.
3. For each point left in the set, assign it to the new canopy if its distance to the first point of the canopy is less than the loose distance T_1.
4. If the distance of the point is additionally less than the tight distance T_2, remove it from the original set.
5. Repeat from step 2 until there are no more data points in the set to cluster.
6. These relatively cheaply clustered canopies can be sub-clustered using a more expensive but accurate algorithm.

An important note is that individual data points may be part of several canopies. As an additional speed-up, an approximate and fast distance metric can be used for 3, where a more accurate and slow distance metric can be used for step 4.

Clustering with MapReduce

Classification Algorithms

Decision Trees

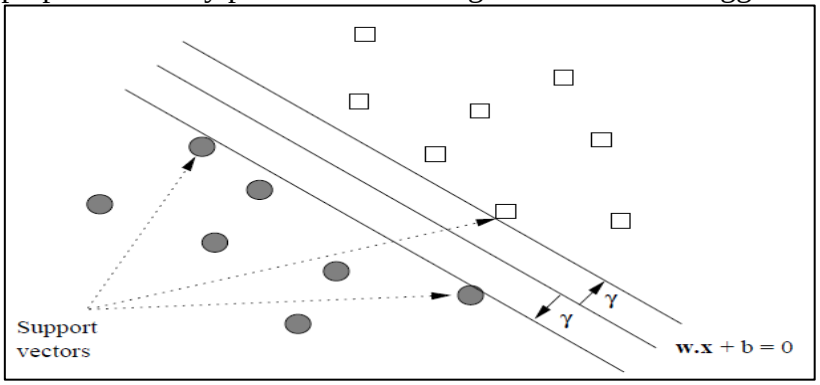
A decision tree is a collection of nodes, arranged as a binary tree. The leaves render decisions; in our case, the decision would be "likes" or "doesn't like." Each interior node is a condition on the objects being classified; in our case the condition would be a predicate involving one or more features of an item.

To classify an item, we start at the root, and apply the predicate at the root to the item. If the predicate is true, go to the left child, and if it is false, go to the right child. Then repeat the same process at the node visited, until a leaf is reached. That leaf classifies the item as liked or not.

Overview SVM Classifiers

An SVM selects one particular hyperplane that not only separates the points in the two classes, but does so in a way that maximizes the margin – the distance between the hyperplane and the closest points of the training set.

The goal of an SVM is to select a hyperplane w.x + b = 01 that maximizes the distance γ between the hyperplane and any point of the training set. The idea is suggested as follows.



There, we see the points of two classes and a hyperplane dividing them. Intuitively, we are more certain of the class of points that are far from the separating hyperplane than we are of points near to that hyperplane. Thus, it is desirable that all the training points be as far from the hyperplane as possible.

Parallel SVM

One approach to parallelism for SVM is you can start with the current w and b, and in parallel do several iterations based on each training example. Then average the changes for each of the examples to create a new w and b. If we distribute w and b to each mapper, then the Map tasks can do as many iterations as we wish to do in one round, and we need use the Reduce tasks only to average the results. One iteration of MapReduce is needed for each round.

A second approach is the contribution from each training example can then be summed. This approach requires one round of MapReduce for each iteration of gradient descent.

One Nearest Neighbor

The simplest cases of nearest-neighbor learning are when we choose only the one neighbor that is nearest the query example. In that case, there is no use for weighting the neighbors, so the kernel function is omitted. There is also typically only one possible choice for the labeling function: take the label of the query to be the same as the label of the nearest neighbor.