Clustering big data

Dan Filimon

June 24, 2013

Contents

1 Rationale

Telescopes surveying the Universe in search of exoplanets, DNA sequencers decoding our genes, governments monitoring cities, and people all over the world exchanging messages, images, videos produce across the Internet produce exabytes of data every day.

Researchers, businesses and governments all over the world are trying to make sense of the ever-increasing collection of data they have access to.

It needs to be stored, searched through, processed and analyzed to extract valuable information. Such a large volume of data comes with its own set of challenges and in the past few years, novel programming paradigms have emerged that make analyzing this data a lot more accessible. There are many systems that are designed to handle “big data”. Out of these, the most famous one is MapReduce [9] from Google, for batch processing and more recently Storm [16] originally from Backtype, now acquired by Twitter.

Many applications that have previously been impossible are now a reality thanks to massive data sets. For example, quality statistical machine Translation can only work at very large scale. While the accuracy of currently available commerical fails to capture more subtle nuances, it is certainly good enough to convey the general meaning.

Large scale data analysis is a broad field with many overlapping disciplines like machine learning, data mining, data science etc.

Analyzing big data is challening. The complexity of many algorithms makes them difficult to parallelize and even when this is possible, because of the scale of the data, clusters of many machines are needed to process the data in reasonable time.

The mainframe era of the 80s, displaced by the affordable personal computer is making a comeback under the guise of “cloud computing” precisely because the complexity of maintaining a large enough cluster of machines to process “big data” makes it a ripe target for outsourcing.

Companies now offer “clouds”, clusters of virtual machines that support MapReduce style processing through open-source Hadoop distributions like Amazon (Amazon Web Services), Google (Google Compute Engine), Microsoft (Windows Azure) have vast data centers at their disposal and offer infrastructure as a service to be billed by the hour (or minute).

Time is quite literally money in this market which makes fast processing even more critical. When processing data at terrabye scale and beyond especially in pipelines, the first priority is usually to reduce the data through some aggregation such that the resulting size is much smaller, for example storing statistics instead of the raw data.

One of the most useful algorithms to analyze data with is clustering. Intuitively the aim of clustering is to find clumps of data that are more similar to one another than the others.

The traditional use of clustering is for revealing interesting patterns in the data — for example, if n good quality clusters do form, this could mean there are n types of users in the system.

It's also especially useful for getting a good representative sample of the data that should behave similarly to the original distribution. This is important because sometimes there are no obvious ways of aggregating the data being processed directly to reduce its size. If the data is clustered however, the resulting clusters' centroids can be used as input for the next stage of processing.

2 Apache Mahout

This work focuses on Apache Mahout [1], an open-source scalable machine learning library written on top of Hadoop [10]. The work I have done is implementing the algorithms described and committing them in Mahout (becoming a committer in the process) where they will be part of release 0.8. The actual code will be described in a later section.

3 Clustering

There are many kinds of clustering which vary significantly in their approach in building the clusters. The kind of clustering we'll be discussing in this paper is "centroid" based.

To apply any clustering, or machine learning for that matter, the data needs to be transformed from its raw representation to "feature vectors". The corresponding feature vector for a data item should capture plausibly relevant characteristics of the item in a numeric form. For example, when working with text documents, feature vectors containing frequencies a given word in a document compared to the corpus are useful.

This paper doesn't attempt to further describe the various ways used to encode data into feature vectors or assess the usefulness of a subset of features. This is requires domain-specific knowledge and is the goal of feature engineering. It's assumed the data is already available as feature vectors and so now the clustering problem can be formalized.

Given n vectors (also referred to as a points) in $\mathbf{R}^d$ and an integer k, and a distance measure, $dist$, group the n points into k disjoint sets X_1 through X_k. The mean of the points in cluster i (or, disjoint set X_i) is called the "centroid" of that cluster, which we call c_i.

The clustering needs to be good in some sense, so define a measure of quality to optimize for. For this, $dist$, a distance measure is needed, making the combined vector space and measure a metric space.

T_c, is what gets optimized for — the total cost of the clustering, which is the sum of the distances from each point to the centroid of the cluter it is assigned to.

$$T_c = \sum_{i=1}^{k} \left(\sum_{\mathbf{x_{ij}} \in \mathbf{X_i}} dist(\mathbf{x_{ij}}, \mathbf{c_i}) \right) \tag{1}$$

In this formulation, the number of clusters, k is fixed, but this problem is related to the facility placement problem, which essentially identical except in that the number of cluster can vary.

The main issue with this formulation is that the optimization problem we're trying to solve is NP-hard in the general case. This has the very important implication that it is infeasible to find an optimal solution to this problem. Any polynomial-time algorithm we can devise will at best be an approximation scheme. This turns out to not be as dire as it first sounds, because "good enough" is fine for virtually all applications and also because real data tends to have additional properties that results in stronger quality guarantees.

3.0.1 Distance measure

One difference users used to clustering literature might have noticed in this formulation is that our choice of distance measure is not fixed. Normally most papers on clustering work with $||\mathbf{x} - \mathbf{y}||_2^2$, the squared Euclidean distance (or squared $L2$-norm). We acknowledge this and indeed some results we use for our implementations have this assumption. As implementors of a machine learning library however, we need to support user extensibility and we feel that other than providing documentation to our users, we shouldn't enforce other kinds of restrictions. It turns out that most distances used in practice are variants of this distance measure (for example the $L2$-norm and the cosine distance), in which case the results likely the same. If some completely different distance measure is used, we can't provide any guarantees.

3.1 Quality

When it comes to the quality of the clusters generated by any algorithm, it is often said that "clustering is in the eye of the beholder". In unsupervised learning, which this problem is a part of, there is no known "right answer". Compare this with classification or regression where an example is either correctly clasified or predicted, or it isn't.

Things are not so clear-cut in this case. While the total cost is certainly what we optimize for, multiple measure have been devised over the years that attempt to formalize the degree a clustering is "good". Intuitively, we want clusters that are:

- compact, meaning that the points in a cluster are close together (the intra-cluster distances are small between any two points)
- well separated, meaning that two different clusters will be relatively far apart (the inter-cluster distances are large between any two clusters)

The Dunn Index and Davies-Bouldin Index try to express compactness and separability in one score. These are called "internal" scores because they only at one clustering.

Additionally, it is often useful to compare two different clusterings and to see how similar they are. This is useful especially when comparing different clusteirng algorithms. A widely used score for this is the Adjusted Rand Index based off the "confusion matrix" (same idea as with classification).

3.1.1 Dunn Index

The Dunn Index, was invented in 1974 by J. Dunn. A higher Dunn Index indicates a better clustering. It combines Δ_i, a distance score for cluster i (called intracluster distance) and the distance between two clusters, $dist(\mathbf{c}_i, \mathbf{c}_j)$ (intercluster distance).

Δ_i can have different expressions. For cluster X_i it could be:

- the maximum distance between any two points

$$\Delta_i = \max_{\mathbf{x},\mathbf{y} \in X_i} dist(\mathbf{x}, \mathbf{y}) \tag{2}$$

- the mean distance between any two points

$$\Delta_i = \frac{1}{|X_i|(|X_i| - 1)} \sum_{\mathbf{x},\mathbf{y} \in X_i, \mathbf{x} \neq \mathbf{y}} dist(\mathbf{x}, \mathbf{y}) \tag{3}$$

- the mean distance between any point and the centroid

$$\Delta_i = \frac{\sum_{\mathbf{x} \in X_i} dist(\mathbf{x}, \mathbf{c}_i)}{|X_i|} \tag{4}$$

- the median distance between any point and the centroid. This is the one we use in the implementation. The reasoning is that the median is much more roubst to outliers than the mean or max. Additionally, computing distances between all pairs of points is simply not feasible for large datasets.

Having defined Δ_i, the Dunn Index is:

$$D = \min_{1 \leq i \leq k} \left\{ \min_{1 \leq j \leq k, j \neq i} \left\{ \frac{dist(\mathbf{c}_i, \mathbf{c}_j)}{\max_{1 \leq l \leq k} \Delta_l} \right\} \right\} \tag{5}$$

3.1.2 Davies-Bouldin Index

The Davies-Bouldin Index was invented in 1979 by D. Davies and D. Bouldin. Like the Dunn Index, it is an internal evaluation scheme. A lower Davies-Bouldin Index indicates a better clustering.

Defining a cluster-specific measure, exactly like for the Dunn Index, Δ_i, the index is:

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\Delta_i + \Delta_j}{dist(\mathbf{c}_i, \mathbf{c}_j)} \right) \tag{6}$$

3.1.3 Adjusted Rand Index

The Rand Index was invented by W. Rand in 1971. It measures how similar two clusterings are to one another. When the index is adjusted for change grouping of elements it is known as the Adjusted Rand Index.

To compute the index, one must first construct a contingency table (also known as a confusion matrix). Assuming the clusterings are $X = \{X_1, X_2, \dots X_k\}$ and $Y = \{Y_1, Y_2, \dots Y_k\}$, the overlap between cluster i of X, and cluster j of Y, n_{ij} is the number of points that are closest to cluster i in clustering X and cluster j in clustering Y. That is $n_{ij} = X_i \cap Y_j$.

	Y_1	Y_2	$\dots$	Y_k	Sums
X_1	n_{11}	n_{12}	$\dots$	n_{1k}	a_1
X_2	n_{21}	n_{22}	$\dots$	n_{2k}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_k	n_{k1}	n_{k2}	$\dots$	n_{kk}	a_k
Sums	b_1	b_2	$\dots$	b_k	

The Adjusted Rand Index is:

$$AR = \frac{\sum_{ij}\binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] / \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] / \binom{n}{2}} \tag{7}$$

4 k-means

The most famous, simple and quite venerable clustering algorithm, known since the 50s is k-means and later Lloyd's method.

It first starts by chosing k point out of the n as seeds. These will be the first centroids (they're not "really" centroids since they are not the means of the points). The algorithm then proceeds by doing some number of iterations of the following steps:

1. assign each point to the cluster whose centroid is closest to it
2. recompute the existing centroids

The following 6 figures illustrates how this works in a toy example with 2D points we want to cluster into 3 clusters.

Figure 1: These are some example points we'll cluster in 3 groups

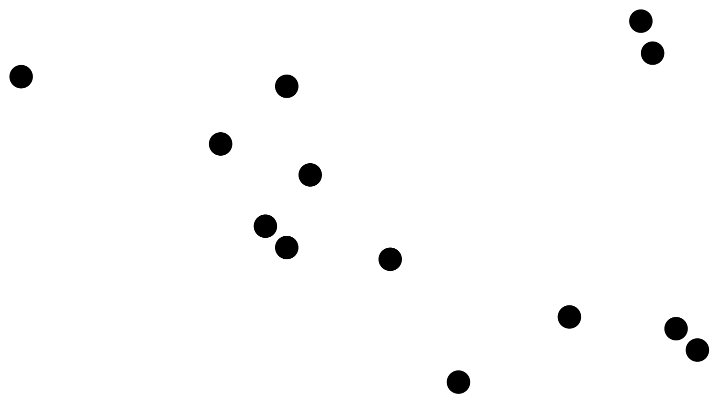


Figure 2: Here we selected 3 points as the initial centroids

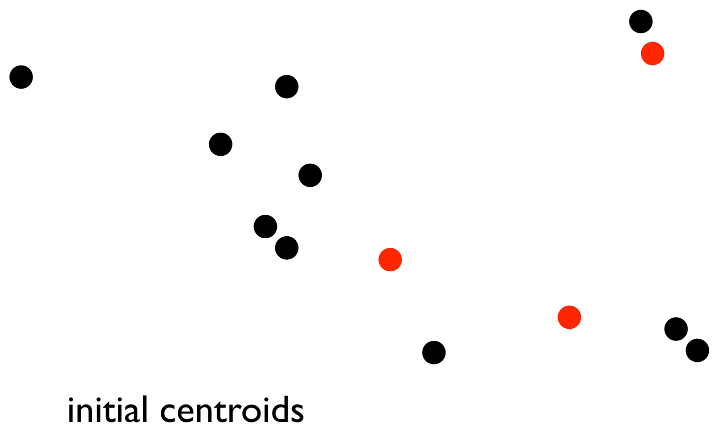


Figure 3: Here we assign each point to the cluster whose centroid it's closest to

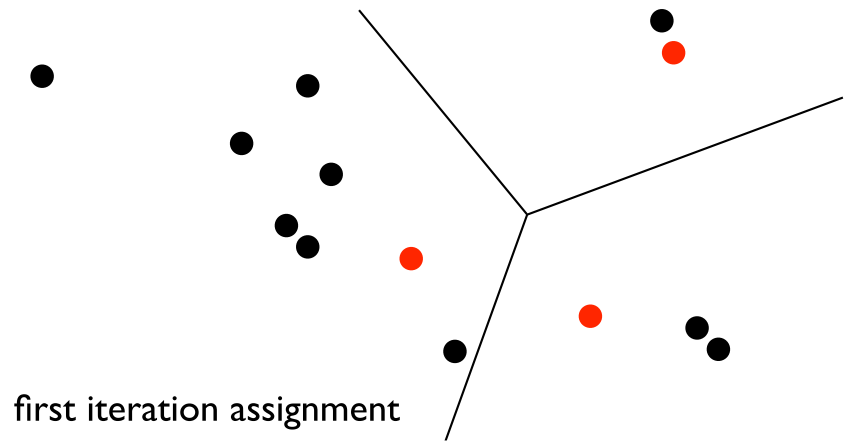


Figure 4: After asigning each point to a cluster, we compute the centroids of those clusters as the mean of the points in that cluster

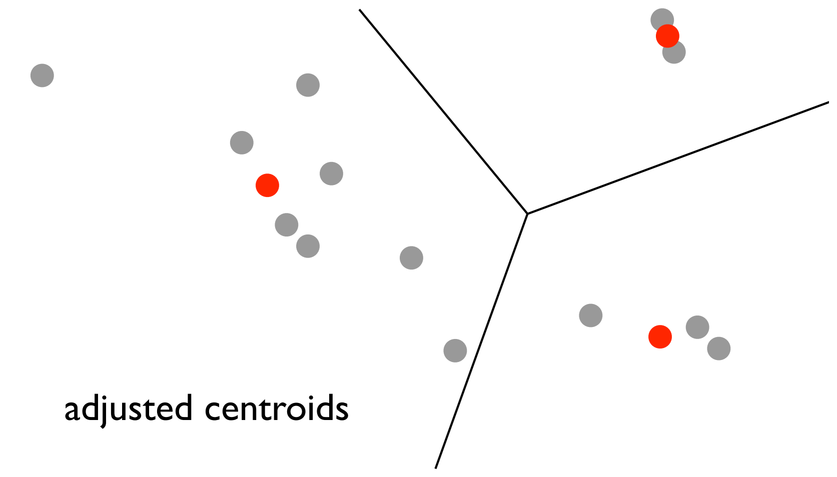


Figure 5: Second iteration assignment

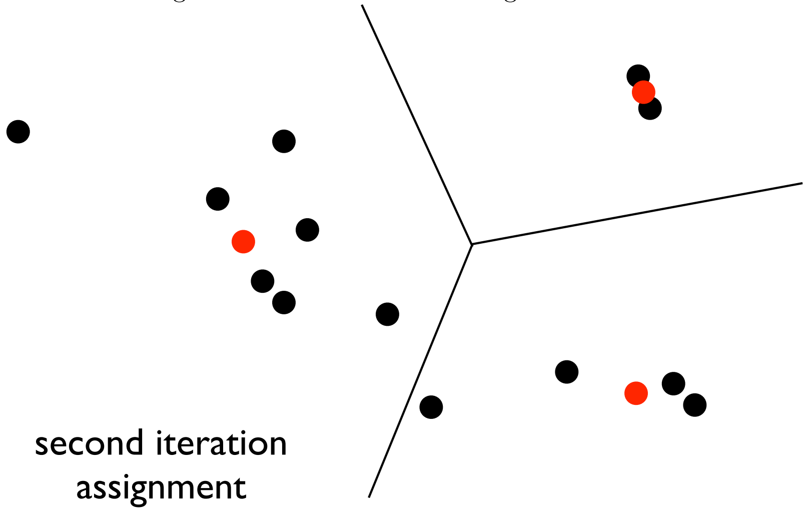
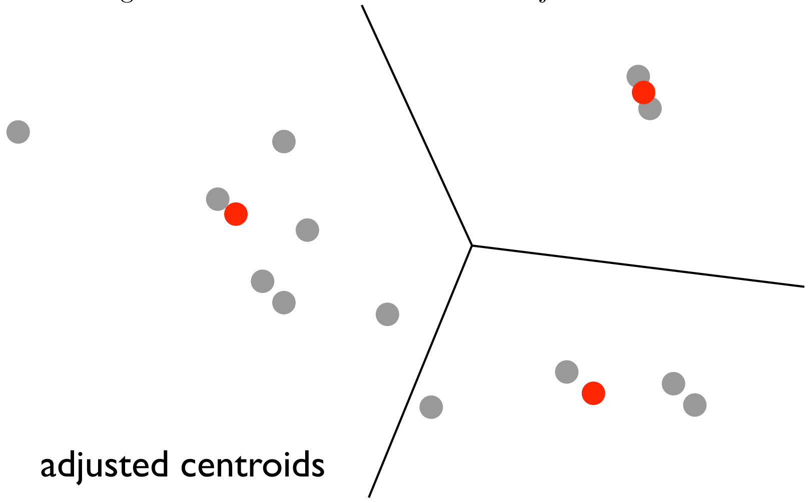


Figure 6: Second iteration centroid adjustment



4.1 Algorithm

The pseudocode for the algorithm described above is:

```
$Centroids \leftarrow select\ k\ points \in Points$
while not done do
    for $c \in Centroids$ do
        $Clusters[c] \leftarrow \emptyset$
    end for
    for $p \in Points$ do
        $d_{min} \leftarrow \infty$
        for $c \in Centroids$ do
            $d \leftarrow dist(p, c)$
            if $d < d_{min}$ then
                $d_{min} \leftarrow d$
                $c_{min} \leftarrow c$
            end if
        end for
        $Clusters[c_{min}] \leftarrow Clusters[c_{min}] \cup \{p\}$
    end for
    $Centroids \leftarrow \emptyset$
    for $C \in Clusters$ do
        $Centroids \leftarrow Centroids \cup \{mean(C)\}$
    end for
end while
```

The crucial details in the algorithm are:

1. centroid initialization: how are the k points selected to become centroids at the beginning?

2. stopping condition: how long do we need to do the point assignment and centroid adjustment?

These two questions have a variety of solutions each of them resulting in a slightly different version of the algorithm, but with mostly similary results.

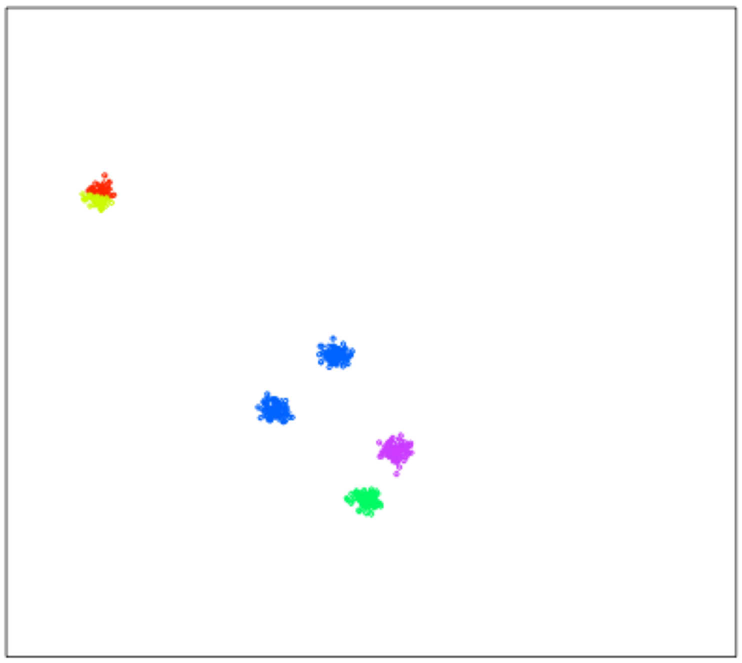
4.2 Initialization

Selecting the first k points to become centroids, or "seeding" as it is also known is the single most important factor in getting a high-quality clustering.

4.2.1 Random selection

The simplest solution is to simply select k points of the n randomly. This produces a seed but has no real guarantees about what these points are. For a good quality clustering, assuming there are k real clusters and we wish to identify them, the seeds should ideally come from the k clusters and no two seeds should come from the same clusters. If this is true, because we assumed the clusters really do exist in the data, the cluster assignment will set nearly all the points in the right cluster in just a few steps. The downside of this approach is illustrated in the figure below. There, two seeds are selected from the same real cluster, causing it to be split among the two k-means clusters (red and yellow). This means, another cluster (the blue one) will have to contain points from two real clusters.

Figure 7: Here is a case where the random initialization failed and a single real cluster was split between the yellow and red k-means clusters



Additionally, like many machine learning algorithms that only result in locally optimal solutions, k-means benefits from multiple restarts. Multiple restarts are even more important as this makes it nearly impossible to not have a good set of initial centroids.

4.2.2 k-means++

There are ways of improving the selection of centroids through the intuitive idea that we want to have seeds that are as far apart to each other as possible. This effectively eliminates the problem of having two seeds in the same cluster. While there are multiple ways of doing this, the most widely used one probably being [2]. The approach I implemented is described in [3] and is summarized below. In [3], Ostrovsky et. all introduce a new condition, called *ϵ-separability* that formalizes what data should look like for a good clustering to exist and give an new way of sampling the initial centroids with provably-good guarantees. This sampling is fairly similar to k-means++ and they show how it can produce a constant-factor approximation of the optimal clustering with just one Lloyd step provided the data is ϵ-separable.

From an implementor's perspective, checking if the data is indeed ϵ-separable is of little interest. We would need to know the optimum clustering cost for k clusters and $k-1$ clusters and if we could compute those, we'd just do it directly. The new sampling method however, is useful and we describe it below (based on section 4.1.1 in [3]).

First, we select two centroids, c_1 and c_2 with probability proportional to $dist(c_1, c_2)$. [3] uses $dist(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2^2$ as the distance measure, but as explained above, we generalize this in the implemetation to any user supplied function. We now have the first 2 centroids.

Then, to select the $(i+1)$-th centroid, call the i already selected centers, $\mathbf{c}_1 \ldots \mathbf{c}_i$. The new point is selected randomly with probability proportional to $\min_{j \in \{1 \ldots i\}} dist(\mathbf{x}, \mathbf{c}_j)$.

4.3 Convergence

As with most iterative machine learning algorithms, the main k-means step is performed multiple times. We stop the algorithm after:

- a fixed number of iterations, this is usually an upper bound
- a quality metric plateaus (in this case, total cost), meaning that it stops decreasing after a few iterations
- no points change cluster assignment

4.4 Number of clusters

Clustering is a very broad and somewhat loosely defined problem in general. In fact, [4] jokingly remarks that "clustering is in the eye of the beholder". While we have formalized the problem in section 2, by explicitly stating that we are given n d-dimensional points and the distance measure *dist*, we also assume to be given k.

This might be true, but most of the time in practice it isn't. Typically however, this being an unsupervised learning problem, there is no ground truth.

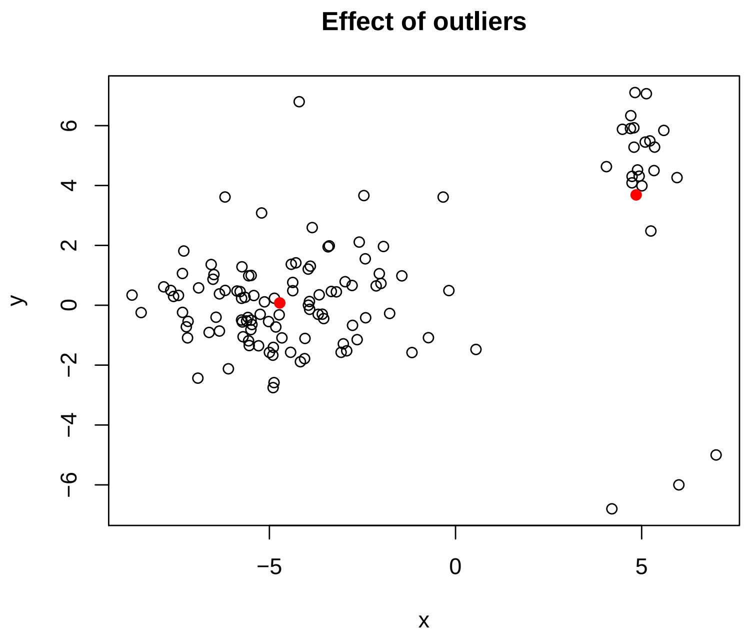
This implies that we can't know what the value of k actually is, and sometimes we don't particularly care (notably, when using clustering to sample our original data set). One well known way of addressing this is the "Elbow Method", where one tries clustering the points with different values of k and plots the total cost T_c as a function of k. The total cost should go down as more clusters are available, because "real" clusters in the data have too few clusters to properly represent them, until after reaching this point, the cost plateaus because more clusters fit the real clusters exactly.

4.5 Outliers

Real data is often messy: it's put together from various data sources by (buggy) algorithms and (falible) humans and errors are likely. These errors look like outliers in the data where one of the dimensions is much different than the rest. In an unlucky case, these outliers migh even be fairly close together, warranting the question of whether they should be thought of as a cluster themeslves.

In the figure below, suppose we're convinced that the 3 points at the bottom right are outliers and that there are only 2 clusters in the data. Then, the reasonable thing to do, is discard them and not have them be part of any cluster.

Figure 8: Data set where outliers have skewed the centroids



That's not possible with standard k-means as every point is part of exactly one cluster and contributes to that cluster's mean. This is why in the figure, the centroid of the cluster in the top-right is skewed downwards towards the three outliers at the bottom-right. The only thing to do here is to manually remove the outliers which is plausible for a small data set, but impossible for even a medium-sized one.

The solution, lifted from [3] modifies the centroid update operation. Instead of taking all the points assigned to a cluster into account when computing the new centroid, onle points that are within a ball (in the topological sense) are averaged. The radius of this ball is fraction (user-configurable in practice) of the smallest inter-cluster distance within the clustering. This ensures that only points close to the cluster's "core" play a role in the adjustment of its centroid.

4.6 Algorithm Complexity

A major consideration not treated up until now in this paper is the complexity of the algorithm. While the number of iterations is highly-dependent on the initial centroid seeding and the particular shape of the data, for a given k-means step, the two main operations take:

1. point to cluster assignment: for each of the n points, go through all k centroids and compute the distance between the d-dimensional vectors. This step is therefore $O(nkd)$.

2. centroid adjustment: the points are partitioned across k clusters, but they are essentially summed up and divided by their count in each cluster. Therefore, this step is $O(nd)$.

So, for a given step, the total cost is $O(nkd + nd) = O(nkd)$. Each of these variables could conceivable be reduced.

1. n: The data could be downsampled, thereby reducing n. Downsampling needs to take into account the particular distribution of the data and preserve it well-enough for the final clustering. This is the route taken indirectly by the streaming k-means approach described in section 9.

2. d: The vectors' dimensionality could be reduced through standard techniques like Principal Component Analysis.

3. k: Searching for the nearest neighbor among all the centroids is usually not a significant expense when k is small. For most "classical" clustering applications, k is less than 100. However, especially when using clustering as a way of approximating data, or when dealing with web-scale data, k can be much larger, potentially on the order of millions. And even if it isn't, being a multiplicative constant is reason enough to want to reduce it. This can be achieved through approximate nearest-neighbor searches.

4.6.1 Approximate nearest-neighbor search

There are many approaches to this problem, some of which are surveyed by Riegger in [6]. The one we chose is based on the Johnson-Lindenstrauss Lemma by W. Johnson and J. Lindenstrauss in [7], for which an elementary proof is given by Dasgupta et. all in [8]. Citing directly from their abstract,

> A result of Johnson and Lindenstrauss shows that a set of n points in high dimensional Euclidean space can be mapped into an $O(\log \frac{n}{\epsilon})$-dimensional Euclidean space such that the distance between any two points changes by only a factor of $(1 \pm \epsilon)$.

The mapping can be obtained by simply sampling $d_r = O(\log \frac{n}{\epsilon})$ d-dimensional random vectors from $\mathcal{N}(0, 1)$ and then normalizing each vector [8].

Arranging these d_r vectors into a $d_r \times d$ sized projection matrix, P, multiplying A by a d-dimensional vector $\mathbf{v}$ results in a d_r-dimensional vector, $P\mathbf{v}$. This is random-projection based dimensionality reduction.

The main problem with nearest-neighbor searches is their inability to narrow-down the candidate set. On scalar data, the two main approaches for fast searching are based either on a total ordering of the elements and binary-search like approaches or a hashing function and use a hash-table.

Vectors in $\mathbf{R}^d$ are not intuitively comparable, and defining an order of the dimensions doesn't work well. To see why, consider points $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbf{R}^d$ and the L1 norm of the difference of two vectors as the distance measure. If the order of the dimensions is from 1 to d, the ordering of the vectors is equivalent to lexicographic. Suppose $\mathbf{b} = [b_1, \ldots, b_d]^T = [a_1 + \epsilon, \ldots, a_{d-1} + \epsilon, a_d + \delta_b]^T$ and $\mathbf{c} = [c_1, \ldots, c_d]^T = [a_1 + 2\epsilon, a_2, a_3 + \epsilon \ldots a_{d-1} + \epsilon, a_d + \delta_c]$. Under the chosen distance measure, $||\mathbf{a} - \mathbf{b}||_1 = (d-1)\epsilon + \delta_b$ and $||\mathbf{a} - \mathbf{c}||_1 = (d-1)\epsilon + \delta_c$. In this case the first $d-1$ dimensions are not relevant to the comparison, yet, in under the ordering $\mathbf{b}$ would come before $\mathbf{c}$ and worse, according to the chosen distance measure $\mathbf{c}$ could even be closer to $\mathbf{a}$ (depending on δ_b compared to δ_c). This shows that this type of ordering, doesn't guarantee the ordering of distances, making it useless for nearest-neighbor searches.

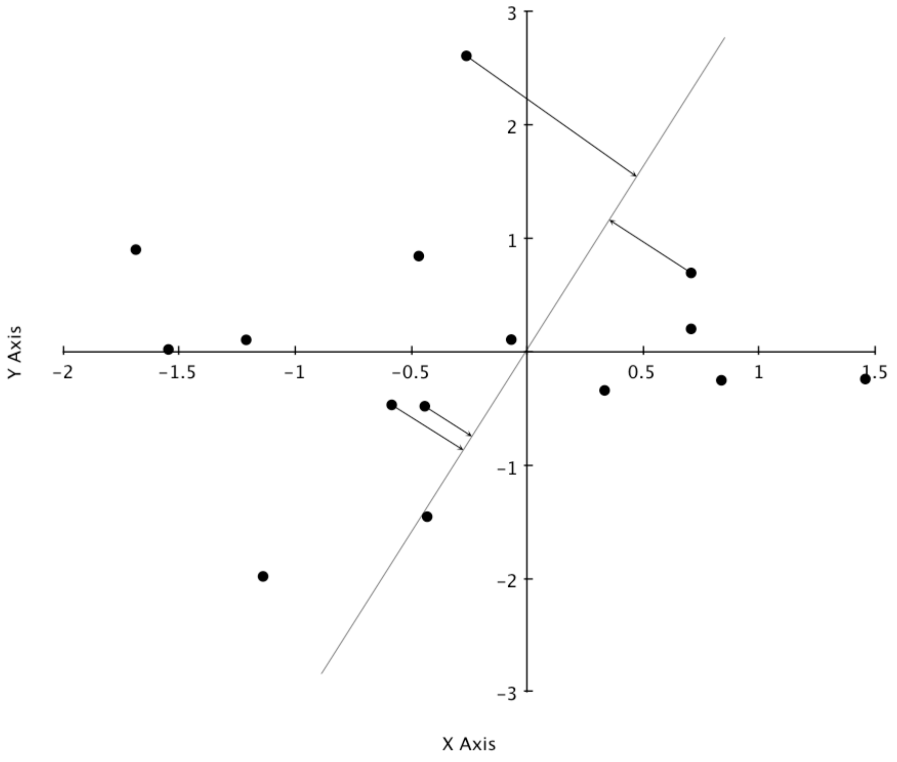
Hashing a vector would result in a scalar that's close to uniformly distributed in some interval so that the resulting hash table is as close to balanced as possible. The issue here is that hashing two similar vectors with a typical good hashing function results in them being far away from each other. This makes trying out more than 1 candidate infeasible and therefore k-nearest neighbor searches impossible.

Random projections offer a better approach. Consider figure 9, below. Call the line such a projection vector, $\mathbf{p}$. The line is the direction of the projection vector, not the vector per se (the vector is normalized to unit length). Each dot then, represents another vector in 2D space. Projecting a vector $\mathbf{a}$ on to another vector $\mathbf{p}$, visualized here as projecting a point on to a line, results in a vector in the same direction as $\mathbf{p}$ and with magnitude $|\mathbf{a}||\mathbf{p}| \cos \angle(\mathbf{a}, \mathbf{p})$. Since all the resulting vectors are in the same direction, just the magnitudes (that are scalar!) need to be compared.

Note that vectors that are close in the original space, will result in comparable magnitudes. This is the very fact that enables the Johnson-Lindenstrauss lemma. However, vectors that are far away in the original space, could also map to similar magnitudes (like the top most 2 projections in figure 9). This means that closeness in the original space guranatees close scalars, but the converse is not true. There could even be points that are farther away from a point in the original space that project more closely to a it than points that are closer to the said point.

This uncertainty gives rise to the approximate nature of the search. The closest neighbor is likely close by, but if the direction of the projection is particularly unlucky, it can be completely ignored in favor of points that are farther away.

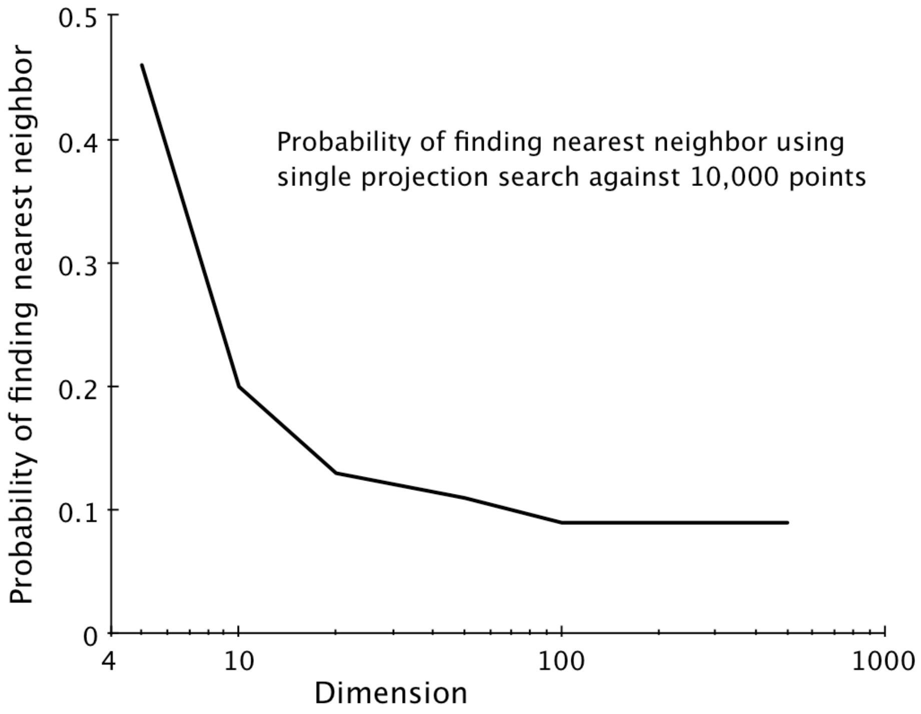
Figure 9: An example of the resulting projections on to a random vector



In fact the approximation becomes better as more directions are considered and the Johnson-Lindenstrauss lemma gives explicit bounds for the number of projections required for required given quality of the answer.

In figure 10, even when using 1 projection, for low-dimensional spaces, the probability of finding the best neighbor directly is quite substantial.

Figure 10:



For the closest k neighbors the strategy now becomes selecting a ball of size b around where the projected query is for each of the p projections, merging all of these balls into a candidate set of size $2bp$ and searching in this new set.

This new set only has $2bp$ points in it which should be a lot less than the original number of vectors to search through. The distance here are computed completely. The size of the ball, b and the number of projections p are the parameters.

```
1: $Candidates \leftarrow \emptyset$
2: for $\mathbf{p} \in Projections$ do
3:     $\mathbf{pq} \leftarrow$ project query on to $\mathbf{p}$
4:     $\mathbf{c} \leftarrow$ closest neighbor of $\mathbf{pq}$ induced by $\mathbf{p}$
5:     $Candidates \leftarrow Candidates \cup$ ball of size $b$ around $\mathbf{c}$
6: end for
7: keep closest $k$ from $Candidates$ by computing the actual distances
```

This approach is implemented as `ProjectionSearch` in Mahout.

5 Large scale

As widely explained in the rationale, large scale data processing is a requirement and clustering is a particularly useful application. There are multiple paradigms that are available, each suitable for particular kinds of problems.

Processing large scale data is such an important problem that there are today many other technologies at Google and in the open-source ecosystem. There are multiple Hadoop distributions, like Mapr [15] that provide complete many tools for both batch and online processing, NoSQL solutions, alternative distributed file systems, support for machine learning etc.

5.1 Batch processing: MapReduce

The most famous approach, popularized by Google's J. Dean and S. Ghemawat in [9], is MapReduce. MapReduce and its open-source Hadoop implementation, [10] are designed for *batch data processing*. The idea is that there is a very large data set to process, that is stable (practically immutable).

This data is typically stored on a distributed file system like Google File System [11] or its open-source HDFS [12] equivalent. The most basic abstraction is the SSTable, or string-string table [11]. It contains records that map serialized keys to values and is referred to in the Hadoop ecosystem as a SequenceFile.

On top of these there can be a NoSQL database layer, like Google BigTable [13] or the equivalent HBase [14]. The database layer offers some structure to the data in the form of a multi-dimensional map [13], section 2.

The best explanation on MapReduce is the original paper [9] and readers unfamiliar with MapReduce are referred to it. A basic understanding of the paradigm is assumed in the rest of this work.

5.2 Online processing: Storm

For particular applications, where there is a continous stream of data that needs to be processed, collecting the data in batches and processing it with Hadoop is challenging. Take Twitter as an example. This widely-used social network with its condensed 140-character messages, and notably hashtags is a great way of understanding trends. Some of these trends are responses to real-world events, like the 2012 US Presidential Election while some are spontaneous and transient.

Batch processing would fail to discern all but the longest-running trends. Twitter's Storm is open-source and designed for online processing. The code and a fairly complete Wiki are avaiable on github [16].

In this approach, the stream of data is analyzed immediately as it becomes available. In Storm lingo, there are:

- Topology: the DAG of nodes used for processing data in Storm. A topology is designed for each type of problem to be solved.
- Tuples: fundamental unit of data that gets passed around through the topology.

- Streams: series of tuples (potentially infinite) that are processed.
- Spouts: sources of streams. This is how data comes in from the outside in a Storm system.
- Bolts: processing nodes, where various kinds of operations are applied to tuples.

While Storm is said to be fault-tolerant [16], what this really means is that if nodes in the topology fail, they will be restarted. For this reason, the operations that are well suited to tuples are stateless. Of course, certain bolts will need to maintain state, if at least for the aggregate statistics that need to be computed. This is left in the hands of the developer and can be implemented as periodic writes to disk of the bolt's state. Storm provides *at least once* guarantees for each processed tuple. So, each tuple is guaranteed to be proceseed. However in the event of a failure, the tuple might be sent through the topology again.

Trident [16] is a higher-level abstraction on top of Storm. It works with concepts like tuple filtering and aggregation which make the system feel more functional, but the key addition is the grouping of the stream in batches. These batches are designed to work as atomic units of computation, the output of a pipeline often ending as a bulk insert into a database. Moreover the batches provide *exactly once* semantics. This is done by assigning each batch a transaction ID and applying state updates on a batch only after the previous batch is fully processed. Interested readers are encouraged to read the full Storm and Trident documentation at [16].

As an interesting note, this system is reminiscent of FlumeJava [17], a higher-level system for implementing MapReduce pipelines developed by Google that also behaves like a Trident-like topology. The main difference is that Trident is designed for online processing, not complex batch pipelines.

6 k-means as a MapReduce

As k-means cannot be expressed as an online algorithm (it simply needs all the points right from the start), the typical large scale implementation uses MapReduce.

6.1 Algorithm

To get a sense of how k-means would work in this setting, let's revisit the pseudocode in section 3.1:

```
1: $Centroids \leftarrow select\ k\ points \in Points$
2: while not done do
3:     for $c \in Centroids$ do
4:         $Clusters[c] \leftarrow \emptyset$
5:     end for
6:     for $p \in Points$ do
```

```
 7:         $d_{min} \leftarrow \infty$
 8:         for $c \in Centroids$ do
 9:             $d \leftarrow dist(p, c)$
10:             if $d < d_{min}$ then
11:                 $d_{min} \leftarrow d$
12:                 $c_{min} \leftarrow c$
13:             end if
14:         end for
15:         $Clusters[c_{min}] \leftarrow Clusters[c_{min}] \cup \{p\}$
16:     end for
17:     $Centroids \leftarrow \emptyset$
18:     for $C \in Clusters$ do
19:         $Centroids \leftarrow Centroids \cup \{mean(C)\}$
20:     end for
21: end while
```

The first question that arises comes from the sampling at line 1. k centroids have to be selected (without replacement) from the n points with equal proability in the simplest case. This is easy to do in memory, but for on-disk data, this is nontrivial. Simply adapting the algorithm for in memory sampling is infeasible because it would require (efficient) constant time access to anywhere on disk.

The solution is using a *reservoir sampling* algorithm, designed to sample points from a stream with equal probability. J. Vitter introduces a family of such algorithms in [18] and compares their performance. For a more informal introduction see this blog post by G. Grothaus [19]. Such algorithms work by passing through the data just once. This is the approach Mahout takes.

In the main part of the algorithm, the *k-means step* is executed multiple times until convergence:

1. Assign each point to its closest cluster;

2. Adjust all the centroids.

Ideally, we'd like to parallelize the outer loop among the mappers and reducers. However, this is impossible because of the intrisinc data dependency in the algorithm. Step i cannot start without having the data from step $i - 1$.

Looking inside the loop, the k-means step is neatly divided into two substeps, as described above. These can be exactly mapped to the mappers and reducers. A very high level description is:

```
1: Sample $k$ centroids from the $n$ points using Reservoir Sampling
2: while not done do
3:     Split the $n$ points across the mappers and for each split, assign the points
       to the $k$ centroids (computing partial sums)
4:     Aggregate the partial sums from the $k$ centroids from every mapper into
       final sums and divide by the number of points per cluster
5: end while
```

6.2 Discussion

This approach is implemented in most of Mahout's clustering algorithms, through a sophisticated and pluggable framework for cluster classification (first step, also given a Bayesian treatment) and various iteration policies (as a MapReduce or in memory). The stop condition is the plateauing of the total cost T_c.

6.2.1 Multiple passes through the data

One unfortunate consequence of the iterative nature of k-means is that for r iterations, there will be $r+1$ passes through the data (the 1 extra pass accounts for the centroid sampling). Sampling the data is not trivial and not parallelizable! Furthermore, the r iterations are actual MapReduce jobs, incurring the penalty of setting up the nodes, moving data through the cluster, etc.

6.2.2 Random initialization

Even more worryingly, the random sampling performed at the beginning wouldn't support the significantly better k-means++ initialization.

While there is more recent work by P. Efraimidis [20] for an algorithm that works with weighted random sampling, it's not clear how this would work for the dynamically adjustable weights required for k-means++ (my guess is that it would not).

The issue with random initialization is that the selected centroids may be paritcularly unlucky, taking many iterations to converge or even failing entirely by selecting two points in the same cluster.

Not only that but the typical solution of multiple random restarts in unfeasible since it requires generating a completely new random sample of centroids and another set of iterations. If the k-means clustering job takes a few hours (easily seen on experimental test sets), getting a good clustering on data would not be practical.

6.2.3 Fast nearest neighbor search

Mahout's k-means also doesn't support fast nearest neighbor searching. This isn't a real theoretical limitation as support for this could easily be added.

7 Making big data small

The k-means as a MapReduce approach described earlier works, but it seems to be a dead end as far as improvement are concerned. A totally different approach, widely used when working with big data is this:

1. Have the data on disk;
2. Construct a sketch of the data that can fit into memory;
3. Apply more sophisticated algorithms on the sketch;

4. See if the results are good for the full data set and maybe adjust the solution accordingly.

8 Streaming k-means

In the streaming setting, the data comes in one point at a time and storing all of the incoming points in memory in infeasible. Algorithms for this setting also exist, notably the work by Ailon et al. in [21]. An improvement of the work in [21] is given by Shindler et al in [22]. This last paper is the source of the Mahout `StreamingKMeans` implementation.

In [22] the number of clusters to generate is not fixed as k as in standard k-mans, but allowed to vary and is on the order of $O(k \log n)$. This makes maintaining the invariant a lot easier.

After all the points in the stream have been processed, the resulting centroids are treatead as a sketch of the original data. This means that each intermediate centroid is a *weighted representative* for the points that were assigned to its cluster by the streaming k-means pass. These weighted points are then clustered in-memory using a traditional ball k-means approach.

8.1 Algorithm

The pseudocode for the streaming k-means algorithm described in [22] and implemented in Mahout is given below. The notable parameters in the algorithm are δ, the distance cutoff, which intuitively corresponds to the maximum distance there can be between a point and the centroid that it's closest to for it to be assimilated. β is the base of the exponential function that drives the increases in distance cutoff before each collapse. It is what insures that collapses will be relatively infrequent.

```
1: for each point $p$ with weight $w$ do
2:     Find the closest centroid $c$ to $p$ and let $d = dist(c, p)$
3:     if even with probability proportional to $d \times w/\delta$ occurs then
4:         create a new cluster with $p$ as its centroid
5:     else
6:         merge $p$ with $c$ updating $c$
7:     end if
8:     if there are more than $O(k \log n)$ clusters then
9:         $\delta \leftarrow \beta \times \delta$
10:        collapse the clusters recursively
11:    end if
12: end for
```

8.2 Discussion

This streaming k-means approach produces a sketch that is provably good-enough for a k-means clustering [21, 22]. Furthermore, given the goal is to create a sketch, approximate versions of nearest-neighbor search are appropriate.

The resulting clustering can be used directly (i.e. for radial basis functions) or another ball k-means pass can collapse the centroids down to k. The important observation here is that the approximation of size $k \log n << n$. For this reason, the resulting data can fit into memory and all the known k-means tricks are applicable. k-means++ initialization, multiple restarts and outlier elimination are all possible.

Therefore the quality of the resulting clustering will be close to that produce by k-means, but with only one (for the most part) pass through the data.

9 Clustering big data: MapReduce

This approach is even more valuable in a MapReduce implementation. Multiple mappers can produce independent sketches of their chunk of the data set that can be merged together as a final set of points.

The mappers will actually overestimate how many centoids are produced. This is because for m mappers and n points, assuming a uniform split, n/m points are to be processed per mapper, resulting in on the order of $k \log(n/m)$ centroids per mapper for a total of $mk \log(n\ m) > k \log n$ and the number of centroids produced increases linearly with the number of mappers. This issue cannot be addressed as it stems from the guarantees provided by each mapper. Fundamentally this means that in large clusters with many mappers and for jobs where the number of clusters is on the order of the number of points, this approach is impractical as it risks producing even more points than the original n. For those cases, different approaches are more appropriate.

Another issue coming from the need for a final ball k-means pass is that of the reduce which must cluster all of the intermediate points. This means that there can be just 1 reducer and that the points from the mappers must fit into memory. This can be remedied by adding another streaming k-means pass in the reducer before the final ball k-means to ensure that the data will fit into memory.

9.1 Mahout implementation

All of the approaches detailed above have been implemented as part of this project in Mahout. The algorithms are now merged into upstream and will be part of the upcoming (as of June 2013) release 0.8.

The main contributions are:

1. `Searcher` & friends: classes and interface that support fast approximate nearest-neighbor search. The most notable implementation are `FastProjectionSearch` and `LocalitySensitiveHashSearch`.

2. `BallKMeans`: implements a classic non MapReduce k-means algorithm supporting k-means++ initialization, multiple restarts with cost evaluations and ball k-means.

3. `StreamingKMeans` & MapReduce classes: the `StreamingKMeans` implements the streaming k-means algorithm described above while `StreamingKMeansMapper`, `StreamingKMeansReducer` and some auxiliary classes implement the Hadoop job.

4. Various other tools for re-splitting Hadoop SequenceFiles, running experiments and measuring cluster qualtiy.

10 Clustering big data: Storm

Storm as outlined above is quite a different paradigm, that's particularly suitable for on-line algorithms like streaming k-means. Since the main focus of this work has been Mahout and implictly Hadoop, the Storm implementation is only a prototype, that serves to highlight how easily `StreamingKMeans` code can be integrated into useful applications.

10.1 Storm prototype

First of all, since the focus of Storm is online processing of streams, talking about batch k-means is unhelpful. As a motivating application, consider anomaly detection. The data could be clustered as it comes in, point by point or by only selecting points ocasionally. The clusters that would result from these two approaches should remain the same over a period of time provided the data doesn't change. If however, points start coming in whose distance to the existing clusters is unexpectedly high making some of the quality metrics change, this indicates that something new is happening and is grouds for triggering an alarm.

A prototype implementation (without the practical application outlined above) is available at [23]. This implementation provides a `StreamingKMeansBolt` as well as a `StreamingKMeansAggregator`. The way streaming k-means clustering works on top of Storm is this:

1. Tuples contain the Vectors that will be clustered.

2. They are clustered directly with `StreamingKMeans.cluster()` as soon as they come in through the regular stream. What results is a continously improving approximation of the input distribution.

3. Periodically (this being implemented in the currenty prototype through Storm `tick tuples`, tuples that are sent through a special stream by the system to every node), the bolt will emit its current approximation for further processing.

11 Results

Now that the approaches and contributions have been adequately presented, it is time for concrete results. First, in terms of expectations, we expect the streaming k-means & ball k-means system to perform thusly compared to the existing k-means:

1. Qualtiy
2. Speed

11.1 Quality

11.1.1 Data sets

11.1.2 Quality measures

11.1.3 Experiment

11.1.4 Discussion

11.2 Speed

11.2.1 Data sets

11.2.2 Cluster

11.2.3 Experiment

11.2.4 Discussion

12 Conclusion

References

[1] Mahout.apache.org Apache Mahout: Scalable machine learning and data mining Mahout.apache.org (2012). Apache Mahout: Scalable machine learning and data mining. [online] Retrieved from: `http://mahout.apache.org/` [Accessed: 20 Jun 2013].

[2] Arthur, D., & Vassilvitskii, S. (2007). k-means++ : The Advantages of Careful Seeding. Proceedings of the eighteenth annual ACM- , 8, 111. Retrieved from http://dl.acm.org/citation.cfm?id=1283383.1283494

[3] Ostrovsky, R. (n.d.). The Effectiveness of Lloyd-type Methods for the k-Means Problem, 118.

[4] Why so many clustering algorithms? - A position paper SIGKDD explorations, Vol. 4, No. 1. (June 2002), pp. 65-75 by Vladimir Estivill-Castro

[5] Indyk, P., & Motwani, R. (n.d.). Approximate Nearest Neighbors : Towards Removing the Curse of Dimensionality, 604613.

[6] Riegger, P. M. (2010). Literature Survey on Nearest Neighbor Search and Search in Graphs, (2300), 136.

[7] W. B. Johnson and J. Lindenstrauss, Extensions of Lipschitz maps into a Hilbert space, Contemp Math 26 (1984), 189206.

[8] Dasgupta, S., & Gupta, A. (2003). An elementary proof of a theorem of Johnson and Lindenstrauss. Random Structures and Algorithms, 22(1), 6065. doi:10.1002/rsa.10073

[9] Dean, J., & Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. Communications of the ACM, 137149. Retrieved from http://dl.acm.org/citation.cfm?id=1327492

[10] Hadoop.apache.org Welcome to Apache Hadoop! Hadoop.apache.org (2011). Welcome to Apache Hadoop!. [online] Retrieved from: `http://hadoop.apache.org/` [Accessed: 18 Jun 2013].

[11] Ghemawat, S., Gobioff, H., & Leung, S. (2003). The Google file system. ACM SIGOPS Operating Systems , 37(5), 29. doi:10.1145/1165389.945450

[12] Hadoop.apache.org HDFS Architecture Guide Hadoop.apache.org (2008). HDFS Architecture Guide. [online] Retrieved from: `http://hadoop.apache.org/docs/stable/hdfs_design.html` [Accessed: 18 Jun 2013].

[13] Chang, F., Dean, J., & Ghemawat, S. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on . Retrieved from http://dl.acm.org/citation.cfm?id=1365816

[14] Hbase.apache.org HBase - Apache HBase Home Hbase.apache.org (2013). HBase - Apache HBase Home. [online] Retrieved from: `http://hbase.apache.org/` [Accessed: 18 Jun 2013].

[15] Mapr.com Apache Hadoop Solutions For Big Data — MapR Mapr.com. Apache Hadoop Solutions For Big Data — MapR. [online] Retrieved from: `http://www.mapr.com/` [Accessed: 18 Jun 2013].

[16] Nathanmarz, N. storm Nathanmarz, N. (2013). storm. [online] Retrieved from: `https://github.com/nathanmarz/storm` [Accessed: 18 Jun 2013].

[17] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., & Weizenbaum, N. (n.d.). FlumeJava : Easy , Efficient Data-Parallel Pipelines, 363375.

[18] Vitter, J. S. (1985). Random sampling with a reservoir. ACM Transactions on Mathematical Software, 11(1), 3757. doi:10.1145/3147.3165

[19] Grothaus, G. Gregable: Reservoir Sampling - Sampling from a stream of elements Grothaus, G. (2007). Gregable: Reservoir Sampling - Sampling from a stream of elements. [online] Retrieved from: `http://gregable.com/2007/10/reservoir-sampling.html` [Accessed: 20 Jun 2013]

[20] Efraimidis, P. (2010). Weighted Random Sampling over Data Streams. arXiv preprint arXiv:1012.0256. Retrieved from `http://arxiv.org/abs/1012.0256`

[21] Ailon, N., Jaiswal, R., & Monteleoni, C. (2009). Streaming k-means approximation. Advances in Neural Information , 19. Retrieved from `http://www1.cs.columbia.edu/~rjaiswal/ajmNIPS09.pdf`

[22] Shindler, M. (2011). Fast and accurate k-means for large datasets. Advances in Neural . Retrieved from `http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract`

[23] Dfilimon, D. streaming-storm Dfilimon, D. (2013). streaming-storm. [online] Retrieved from: `https://github.com/dfilimon/streaming-storm` [Accessed: 24 Jun 2013].