CS 246: Mining Massive Data Sets

Samuel Wong
Department of Statistics
Stanford University

Abstract

The course takes a journey through different techniques of processing big data. The course can be separated into 5 different sections: high dimensional data, graph data, infinite data (streams), machine learning, and applications. The high dimensional data section is covered by explaining locality sensitive hashing, clustering, and dimensionality reduction. The graph data section includes PageRank, SimRank, Network Analysis, and Spam detection. The infinite data section talks about filtering data streams, web advertising, and queries on streams. The machine learning section discusses SVM and decision trees. Lastly, the applications involve recommender systems, association rules, and duplicate document detection. Many of these topics cover not only the theory behind obtaining an exact solution, but also how to obtain an approximate solution in a rougher, but computationally efficient way when working with enormous data sets.

Contents

1 MapReduce and Spark **3**
1.1 Overview . . . 3
1.2 Distributed File System . . . 3
1.3 MapReduce . . . 3

2 Frequent Itemsets Mining **5**
2.1 Definitions . . . 5
2.2 Finding Frequent Itemsets . . . 5
2.3 A-Priori Algorithm . . . 5
2.4 PCY Algorithm . . . 6

3 Finding Similar Items: Locality Sensitive Hashing **7**
3.1 Shingling . . . 7
3.2 Min-Hashing . . . 7
3.3 Locality Sensitive Hashing . . . 8
3.4 LSH for Different Distance Metrics . . . 8

4 Clustering **9**
4.1 Hierarchical Clustering . . . 9
4.2 K-Means . . . 9
4.3 BFR Algorithm . . . 10
4.4 CURE Algorithm . . . 10

5 Dimensionality Reduction **11**
5.1 SVD . . . 11
5.2 CUR . . . 11

6 Recommender Systems **12**
6.1 Content Based Recommender Systems . . . 12
6.2 Collaborative Filtering . . . 12
6.3 Latent Factor Based . . . 13
6.4 Other Ideas . . . 13

7 Analysis of Large Graphs: Link Analysis **14**
7.1 PageRank . . . 14
7.2 Topic-Specific (Personalized) PageRank . . . 15
7.3 TrustRank . . . 15

8 Community Detection in Graphs **16**

9 Graph Representation Learning **18**
9.1 DeepWalk . . . 18
9.2 node2vec . . . 18
9.3 Embedding an Entire Graph . . . 18

10 Large-Scale Machine Learning **19**
10.1 Decision Trees . . . 19
10.1.1 How to split? . . . 19
10.1.2 When to stop splitting? . . . 19
10.1.3 How to predict? . . . 19
10.1.4 Building Decision Trees Using MapReduce . . . 20
10.2 Support Vector Machines . . . 20

11 Mining Data Streams **21**
11.1 Sampling from a Data Stream . . . 21
11.2 Queries Over a Long Sliding Window . . . 21
11.3 Filtering Data Streams . . . 22
11.4 Counting Distinct Elements . . . 22
11.5 Computing Moments . . . 22
11.6 Counting Itemsets . . . 23

12 Advertising on the Web **24**

13 Learning Through Experimentation **25**

14 Optimizing Submodular Functions **26**
14.1 Submodularity . . . 26
14.2 Probabilistic Set Cover . . . 26
14.3 Lazy Evaluation of Submodular Functions . . . 26
14.4 Determining the Concept Weights . . . 27

1 MapReduce and Spark

1.1 Overview

Data mining: the extraction of actionable information from very large datasets

Since the data is very large, we want to distribute the computing to multiple machines. Some challenges that we face:

- How do you distribute computation?
- How can we make it easy to write distributed programs?
- Machine failures

Since copying data over a network takes time, we can instead bring the computation to the data. We can then store files multiple times for reliability. Spark and Hadoop addresses these problems.

1.2 Distributed File System

Typical usage pattern:

- Huge files (100s of GB to TB)
- Data is rarely updated in place
- Reads and appends are common

The system consists of:

- Chunk servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might also be replicated as well
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

In a reliable distributed file system, the data is kept in chunks that are spread across different machines. These chunk servers also serve as compute servers, thereby bringing the computation to the data!

1.3 MapReduce

MapReduce: a style of programming designed for:

- Easy parallel programming
- Invisible management of hardware and software failures
- Easy management of very-large-scale data

Several implementations of MapReduce include Hadoop, Spark, Flink etc. MapReduce consists of 3 different steps:

- Map: maps each key to a value
- Group by Key: groups by key, items with the same keys are placed together with each other on a single server
- Reduce: reduces the value by each key

An example using word count of a large document. In the map step, each word is mapped to (word, 1). Group by key will, for each different word, put all instances of that on the same server. Reduce will them sum up the values of each word pair to get the total count.

The MapReduce environment automatically takes care of partitioning the input data, scheduling the program's execution across a set of machines, performing the group by key step, handling machine failures, and managing required inter-machine communication. All the user has to do is write the map and the reduce functions.

Spark extends MapReduce by having computations in memory using RDD (resilient distributed datasets) and allowing for functions outside of just map and reduce. Spark has lazy evaluation as well as higher level APIs built on top of the RDD (such as Spark DataFrames and SparkSQL)

2 Frequent Itemsets Mining

Goal: Identify items that are bought together by sufficiently many customers. This is also called market basket analysis.

We first define items and baskets. Items are individual things bought by a customer. Baskets are collections of subsets of items. We want to discover association rules (e.g. people who bought {x,y,z} also bought {v,w}).

2.1 Definitions

We start with these definitions:

- Support for itemset I: Number of baskets containing all of the items in I
- Frequent itemsets: Itemsets that appear at least as much as the support threshold (defined by user)
- Association rules: $\{i_1, i_2, ..., i_k\} \rightarrow j$ means that if a basket contains all of $\{i_1, i_2, ..., i_k\}$, then it is likely to contain j
- Confidence of an association rule: $\frac{support(I \cup j)}{support(I)}$. Number of baskets with all of the items in I as well a j, divided by the number of baskets with all of the items in I
- Interest of an association rule: Absolute value difference between the confidence of an association rule and the fraction of baskets that contain j. $Interest(I \rightarrow j) = |conf(I \rightarrow j) - P[j]|$

Our goal is to find all the association rules that have a support over some value s and a confidence over a value c. Computationally, the difficult part is finding the number of frequent itemsets.

2.2 Finding Frequent Itemsets

In order to find all the frequent itemsets, we need to generate all the itemsets and then figure out which ones are frequent. The naive approach to do this (say for pairs), is to generate every combination of items. This approach fails if the $(numberofitems)^2$ exceeds the main memory limit. So how can we do this in a better way?

2.3 A-Priori Algorithm

A two-pass approach that can help us solve this large computation problem by limiting the amount of itemsets generated. How it works is that in the first pass, we count the number of occurrences of each item, and we note that if each individual item is not frequent, then no pair including that item can be a frequent itemset. (For example, if item A does not appear 100 times, itemset with A and B cannot appear 100 times).

In the second pass, we read the baskets again and keep track of the count of only those pairs where both elements are frequent.

We can extend this algorithm when going from pairs to triples, etc.

A step by step example of how this algorithm would work:

- $C_1 = \{\{b\}\{c\}\{j\}\{m\}\{n\}\{p\}\}$
- Count the support of itemsets in C_1
- Prune non-frequent. We get: $L_1 = \{\{b\}\{c\}\{j\}\{m\}\}$
- Generate $C_2 = \{\{b,c\}\{b,j\}\{b,m\}\{c,j\}\{c,m\}\{j,m\}\}$
- Count the support of itemsets in C_2
- Prune non-frequent. $L_2 = \{\{b,m\}\{b,c\}\{c,m\}\{c,j\}\}$
- Generate $C_3 = \{\{b,c,m\}\{b,c,j\}\{b,m,j\}\{c,m,j\}\}$*
- Count the support of itemsets in C_3
- Prune non-frequent. $L_3 = \{\{b,c,m\}\}$

*Note that in C_3, don't actually need to generate the last 3 triples as there are pairs in there that are not frequent.

2.4 PCY Algorithm

An improvement to the A-Priori algorithm by exploiting the unused memory during the first pass. We notice that in the first pass, where we are looking only at singletons (not pairs), we have unused memory that can be used to do something else.

In PCY, we use this unused memory to keep a hashtable with as many buckets as fits into the unused memory. Using these buckets, as we pass through the singletons, we hash pairs of them into the different buckets and save the count. Since the number of buckets will be less than the number of pairs (or else we wouldn't need to do this at all), we will get hash collisions. We note that if a bucket contains a frequent pair, then that bucket is surely frequent. However, even without any frequent pair, a bucket can still be frequent. What we DO know though, is that for a bucket with a count less than the support, definitely none of its pairs are frequent. So what we gain from this algorithm is that for pairs that hash into one of these buckets with a count less than the support, we can eliminate them as candidates completely. This way, when we do the second pass through in A-Priori where we generate pairs, we don't have to generate these pairs because we know that they will not be frequent.

3 Finding Similar Items: Locality Sensitive Hashing

Goal: We want to find similar items (de-duplicate) and group them into sets. A naive way to do this would be to compare the vector representations of every pair of items to see if they are very similar or not, which would take $O(N^2)$, where N is the number of data points. We want to do this in $O(N)$ time instead.

Some use cases of this would be finding similar documents to a query on a search engine, similar products to one a user likes, etc.

In order to find similar documents, we have a 3 step process:

1. Shingling
2. Min-hashing
3. Locality Sensitive Hashing

3.1 Shingling

We use shingling to convert a document into a set representation. We use a k-shingle (or k-gram), which is a sequence of k tokens that appear in the document. For example, a 3-shingle of the sentence "I went to the store" is {{I went to} {went to the} {to the store}}.

Two benefits of using shingles to represent a document are:

- Documents that are intuitively similar will have many shingles in common
- Changing a word only affects k-shingles within distance k-1 from the word

When we compare two documents using shingles, since they are a "set" representation, we want to use Jaccard similarity.

Jaccard similarity: $sim(D_1, D_2) = |\frac{C_1 \cap C_2}{C_1 \cup C_2}|$
Jaccard distance: $1 - sim(D_1, D_2)$

For two documents, we could compare the two sets of shingles created, however, again for large documents (and a large corpus), these vectors will be extremely sparse and therefore computationally expensive (one document could be 1M or 10M long). So we use Minhash to reduce this.

3.2 Min-Hashing

The goal of minhash is to convert large sets to short signatures, while preserving the similarity measure. In other words, when we hash a column C to a hash h(C), we need to find a hash function such that if $sim(C_1, C_2)$ is high, then $h(C_1) = h(C_2)$ and when $sim(C_1, C_2)$ is low, then $h(C_1) \neq h(C_2)$.

Minhash is for Jaccard similarity measurements (comparing sets). We have other hash functions for other similarity metrics. Minhash steps:

- Permute the rows of the Boolean matrix using some permutation π
- Define minhash function for this permutation π, $h_\pi(C)$ = the number of the first row (in permuted order) in which column C has value 1
- Apply, to all columns, several randomly chosen permutations π to create a signature for each column
- Result is a signature matrix: Columns = sets, Rows = minhash values for each permutation π

Basically we have one hash function for each permutation, and we do as many permutations (usually say 100) as we like to get an (length 100) encoding for each document. We have just now reduced the vector representation of each document from a length of (maybe 1M or 10M) to 100. Note that the more hash functions we use, the more accurate our signature matrix will be to the original, but higher computational cost. Vice versa with fewer hash functions.

3.3 Locality Sensitive Hashing

The general idea of locality sensitive hashing is that we use a a hash function that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated.

How this works is we take a signature matrix (provided from the Minhash). This signature matrix will have a column for each document, and a row for each Minhash hash function. We divide this signature matrix into bands of rows. For example, if our signature has N documents and 100 hash functions for a representation, the matrix will be 100 X N. We can split this matrix is 20 bands of 5, so each 5 rows next to each other will be in the same band.

For each pair of documents, for each band, we compare those values in their signature matrix representation, and if all 5 are the same, then they hash into the same bucket. We do this for all 20 bands. If at least one of bands hashes to the same bucket, we declare these two documents a candidate pair.

We can figure out that by doing this algorithm (locality sensitive hashing), we probabilistic-ally bucket similar documents together. However, because it is probabilistic, there will be some false positives (two documents are not similar but hash to the same bucket) and false negatives (two documents are similar but hash to different buckets for all bands). We can adjust how much of each by adjusting our values of r and b (rows per band and number of bands). Looking at a graph, we can see that an S-curve is formed by our choice of r and b, and we can select where the S occurs by our choice of r and b.

Generalizing this idea of locality sensitive hashing, we have the following definition. A family H of hash functions is said to be (d_1, d_2, p_1, p_2)-sensitive if:

- If $d(x, y) \leq d_1$, then the probability over all $h \in H$, that $h(x) = h(y)$ is at least p_1
- If $d(x, y) \geq d_2$, then the probability over all $h \in H$, that $h(x) = h(y)$ is at most p_2

What we realize is that we can do locality sensitive hashing with any (d_1, d_2, p_1, p_2)-sensitive family of hash functions! Then we can amplify our tolerance of false positive and false negatives by creating these bands and rows and choosing b and r.

3.4 LSH for Different Distance Metrics

We can use cosine distance as a distance metric. The hashing function that we use for cosine distance then is called "random hyperplanes." Random hyperplanes is a $(d_1, d_2, 1 - \frac{d_1}{\pi}, 1 - \frac{d_2}{\pi})$-sensitive family of hash functions. How it works is that the two "documents" are represented by some vector. We then select random vectors and take the dot product of this randomly chosen vector with both of the vectors that represent the two documents. If the dot product is greater or equal to 0, then we assign the value +1, otherwise -1.

Intuitively, if those documents are similar, they will point in a very similar direction, and they angle between them will be small. What we are doing is selecting random vectors and seeing if both vectors are on the same side of this random vector, or if the randomly chosen vector is lying in between the two vectors. If both vectors are on the same side of the randomly chosen vector, they will then both be +1 or both be -1. If the vector is lying in between the two vectors, one will hash to +1, and one will be -1, showing a difference.

We can use a technique called "projection onto lines" when we use Euclidean distance as our distance metric. How this works is that we generate random lines in space, and partition them into different buckets of equal size. We take each point (each point is one document) and project it into that line and figure out which bucket it is in. The idea is that points close to each other will likely project onto the same bucket in the line

4 Clustering

Goal: Given a set of points, with a notion of distance between points, we want to group the points into some number of clusters, so that members of the same cluster are close/similar to each other and members of different clusters are dissimilar. Typically, points are in high dimensional space and similarity is defined using a distance measure (Jaccard, Euclidean, cosine etc.)

Clustering can be a difficult problem, because while clustering in two dimensions may be easy, in higher dimensions we run into the curse of dimensionality, so in high dimensions, all the points are far away from each either. We can cluster using a hierarchical approach (bottom up), or a divisive approach (top down).

4.1 Hierarchical Clustering

With hierarchical clustering, we repeatedly combine the two nearest clusters. We run into three important questions:

- How do you represent a cluster of more than one point?
- How do you determine the "nearness" of clusters?
- When to stop combining clusters?

If we are using Euclidean distance, we can represent a cluster of more than one point by using a notion of a centroid (which is the average of the datapoints in that cluster). The "nearness" of the clusters can be calculated as the distance between the centroids of those clusters.

In the non-Euclidean case, we can represent a cluster of more than one point by selecting the existing datapoint in that cluster that is closest to all the others in that cluster, also known as the clustroid. The "nearness" of the clusters dcan be calculated as the distance between the clustroids. Note that a clustroid (the closest point) can have different definitions. For example, "closest" could mean the smallest maximum distance to other points, smallest average distance to other points, or smallest sum of squares of distances to other points.

Another approach to determine the "nearness" between two clusters is to calculate the intercluster distance, defined as the minimum of the distances between any two points, one from each cluster.

A third approach to determine the "nearness" between two clusters is to use a notion of cohesiveness. We want to merge two clusters whose union is the most cohesive when joined. Cohesiveness can be calculated different ways, such as using the diameter of the merged cluster, using the average distance between points in the cluster, or using a density-based approach, such as taking the diameter or avg. distance, and dividing by the number of points in the cluster.
There are different ways we can decide when to stop merging clusters. We can stop merging when some predefined number k clusters is found. We can also stop when some stopping criterion is met (such as if the diameter exceeds some threshold, if the density is below some threshold, or if merging clusters yields a bad cluster.

4.2 K-Means

A popular clustering algorithm. Assumes Euclidean space, and requires a predefined k number of clusters. Clusters are initialized by picking k random points. We can also initialize by picking the first point at random, and then picking the subsequent k-1 points as far as possible from the first point (this will work well if we don't have outliers).

An improved way of initializing these k cluster points is called K-means++. How this works is we first select a number of points S, that is logarithmic to the total number of points. We select these S points by visiting points in a random order, but the probability of adding a point p to the sample is proportional to the distance between this point p and the nearest already selected point in S. That way, points that are already close to other points in the sample S are less likely to be selected. This sample S is then clustered to k, and the centroids of these k clusters are used as the initial k points for the actual clustering for the whole dataset.

After initializing the k points, K-Means clusters by the following steps:

1. For each point, place it in the cluster whose current centroid it is nearest
2. After all points are assigned, update the locations of centroids of the k clusters
3. Reassign all points to their closest centroid

We repeat steps 2 and 3 until we converge. We converge when points don't move between clusters and the centroids stabilize.

An improvement to K-Means to extend to large data is called the BFR algorithm.

4.3 BFR Algorithm

BFR algorithm is a variant of K-Means designed to handle very large data sets. The idea behind BFR is that it will make two passes through the data, the first pass being to find the cluster centroids, and the second pass will just assign points to the clusters. Since the data is too large to keep in memory, rather than keeping points in the first pass, we keep summary statistics of groups of points.

BFR has a notion of 3 different types of sets:

1. Discard set: points that are close enough to a centroid to be summarized and then discarded
2. Compression set: points not near enough to a centroid but are close to other points and are summarized and discarded, but not assigned to an existing cluster
3. Retained set: isolated points waiting to be assigned to a compression set, do not summarize these

How BFR works, step by step:

1. Initialize K clusters/centroids (can use random initialization or k-means++ technique)
2. Load in a bag of points from disk
3. Assign new points to one of the k original clusters, if they are within some distance threshold of the cluster (discard set). Then discard the points and keep the summary statistics.
4. Cluster the remaining points, and create new clusters (Compression set).
5. Try to merge new clusters (compression set) from step 4 with any of the existing clusters (discard set).
6. Repeat steps 2-5 until all points are examined.
7. If this is the last round, merge all compressed set points and all retained set points into their nearest cluster

For each cluster, the set is summarized by 3 things: the number of points N, the vector SUM which has the sum in each dimension for all the points in that cluster, and the vector SUMSQ which has the sum squared in each dimension for all the points in that cluster. Using these three items, we can calculate the average and variance for each cluster.

A couple of implementation details: How do we decide if a point is “close enough” to a cluster that we will add the point to that cluster? How do we decide whether 2 compression sets (CS) deserve to be combined into one? To determine if a point is "close enough" to a cluster to be added, we use the Mahalanobis distance. THe Mahalanobis distance is a normalized Euclidean distance (based on mean and average and assuming Gaussian). We can combine 2 compression sets if the combined variance is below some threshold.

However, with K-Means and BFR, we assume clusters are normally distributed in each dimension. Axes are fixed, for example ellipses at an angle are not ok. Therefore, we can use the CURE Algorithm for arbitrary shapes.

4.4 CURE Algorithm

The CURE (Clustering Using REpresentatives) Algorithm is an extension of K-Means to clusters of arbitrary shapes. CURE is a 2 pass algorithm with the following passes:

1. Pick a random sample of points that fit into main memory. Create initial clusters by clustering these points hierarchically. For each cluster, we pick a sample of points as dispersed as possible (called the representatives). Move each representative 20% toward the centroid of the cluster.
2. Now, rescan the whole dataset and visit each point p in the data set. Place it in the “closest cluster”. We do this by finding the closest representative point to p and assigning p to that representative’s cluster.

The intuition behind doing the 20% move inward is that a large, dispersed cluster will have larger moves from its boundary and a small, dense cluster will have only a small move. Therefore, this favors a small, dense cluster that is near a larger dispersed cluster.

5 Dimensionality Reduction

When we have too many dimensions, it can make computation difficult, so we want to be able to reduce the dimensions without losing too much information about the original vectors.

5.1 SVD

Singular Value Decomposition, by the Eckhart Young theorem, is the best low rank approximation. In other words, when we do $A = U\Sigma V^T$, and reduce the rank and try to reconstruct A, we minimize the reconstruction error.

An example of SVD: Say we have a matrix with users as rows, movies as columns, and what that user rated that movie as the values in the matrix (1 - 5). Doing an SVD of the matrix, $A = U\Sigma V^T$, V is a mapping between movies and "concepts" and U is a mapping between users and "concepts." Σ is the "strength" of each concept.

In order to do dimensionality reduction, we would perform the SVD of the matrix A. The singular values of the matrix Σ can be ordered from the largest to the smallest on the diagonal. We can select the ones that are the largest and set the other singular values to 0, thereby reducing the dimensions. So how many singular values should we choose? A good rule of thumb is if the "energy" of a set of singular values is the sum of their squares, then we select enough singular values to have at least 90% of the energy.

How do we actually compute the SVD? We can do this by looking at A^TA and AA^T and realizing that we actually need to find the eigenvalues and eigenvectors of those two symmetric matrices. How we actually do that can vary (see CME302 notes), but some ways may be through power iteration or orthogonal iteration.

The benefit of SVD is that it is the optimal low-rank approximation in terms of the Frobenius norm. Some drawbacks are that there is a interpretability problem (don't know what the new bases mean) and a lack of sparsity. We may have a very sparse matrix A, but U and V will be dense! We can help this problem by using the CUR Decomposition instead!

5.2 CUR

CUR decomposition solves the sparcity problem (A is sparse, but U, V are not) by using only (randomly chosen) rows and columns of A. Therefore, if A is sparse, then by using those rows and columns, we will have sparse matrices as well.

Similar to SVD, we decompose A into three matrices C, U, R; however, C and R are not orthogonal. In CUR, we create the C matrix by randomly selecting columns of A. We create the R matrix by randomly selecting rows of A. Let us then denote a matrix W as the "intersection" of these sampled columns/rows. U is the pseudoinverse of the matrix W, which will be the actual inverse if W is nonsingular.

So how do we select these rows/columns into our C and R matrix? We denote the "importance" of a row/column by the square of its Frobenius norm. When selecting rows/columns, we pick proportional to its importance. The intuition behind this is that using this methodology, CUR is more likely to pick points away from the origin, which is what we want.

It can be mathematically proven that if we select $c = O(\frac{klogk}{\epsilon^2})$ columns and $r = O(\frac{klogk}{\epsilon^2})$ rows of A, that the CUR error: $||A - CUR||_F \leq (2+\epsilon)||A - A_K||_F$ with probability 98%. Basically we are saying the CUR error most likely is bounded by the SVD error.

The pros of CUR are easy interpretation and a sparse basis. A con is that columns of large norms will be sampled many times (duplicate columns/rows). In practice, CUR is much less computationally expensive, and can beat SVD at low ranks, but if we have infinite compute/storage and move to higher ranks, SVD will eventually overtake CUR.

6 Recommender Systems

Our goal is to be able to recommend items specific to a particular user to that individual in a personalized way.

Traditionally, shelf space was scarce, so would shelve only the popular products. However, as things moved from physical to the web, the "shelf space" of the web is infinite, and coupled with the abundance of products, we need recommendation filters. The most basic methods of recommending just the most popular items, or hand curated ones, were doing ok, but to improve, we want to be able to recommend specific items to each user differently depending on their taste.

We have three approaches to recommender systems: content-based, collaborative filtering, and latent factor based.

6.1 Content Based Recommender Systems

The main idea of a content based recommender system is to recommend items to customer x similar to previous items rated highly by x. Using this methodology, we only look at the past history of customer x, and these recommendations will depend solely on customer x.

How it works is we will look at the items that customer x has rated in the past. For each item, we will create an item profile. This can be a vector of features. For example, if we look at movies, the item profile can be the author, title, actor, director, etc. If we look at documents, the item profile can be a set of "important words" in the document determined by TFIDF.

We then create a user profile, which can be a weighted average of rated item profiles. Then we can predict the rating a user would give an item by taking the cosine similarity between that user profile and the item profile. Doing this with all the users and items could be computationally expensive, so note that we could use Locality Sensitive Hashing here.

The pros of content based recommender systems include:

- No need for data on other users
- Able to recommend to users with unique tastes
- Able to recommend new and unpopular items
- Easy to provide explanations as to why that item was recommended by listing content-features that caused an item to be recommended

The cons of content based recommender systems include:

- Finding the appropriate features is hard
- Recommendations for new users that have not previously rated anything is hard
- Never recommends items outside user's content profile, so can become overspecialized

6.2 Collaborative Filtering

In collaborative filtering, we harness the quality judgments of other users. There are two types of collaborative filtering, user-user and item-item. We can use the Pearson correlation coefficient (but only on items rated by both user x and user y) as a similarity measure.

In user-user collaborative filtering, we look at the items rated by both our user of interest, x, and other users, y. We calculate the Pearson correlation coefficient as a measure of how similar user x is to user y. To predict the rating of a user x on another item, we take the weighted average (based on the correlation coefficient between x and the other users) of the ratings that other users gave to that item. This weighted average is our prediction of what user x would rate that item.

In item-item collaborative filtering, we determine the similarity of two items based upon the different ratings that were given to the items. We then take the weighted average based on these item similarities to produce a prediction for a user x that has not rated that item. Formulaically, we have:

$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} s_{ij} \cdot (r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} s_{ij}}$, where $b_{xi} = \mu + b_x + b_i$ and μ is the overall ratings across everything, b_x is the rating deviation $(r_x - \mu)$ for that user, and b_i is the rating deviation $(r_i - \mu)$ for that item.

In practice, it has been observed that item-item often works better than user-user since items are simpler, and users have multiple tastes that can change over time. The pros of collaborative filtering include that it can work for any item. The cons include that you have a cold start problem because you need enough users in the system to find a match, spacity, and popularity bias.

6.3 Latent Factor Based

Latent factor is another way of saying to use dimensionality reduction using SVD on the matrix of ratings. We can follow the SVD steps and extract the directions with the most significance.

Latent factor based models will have the same pros and cons as SVD, with ease of interpretability, sparcity etc. One other complication to using latent factor based models is that a non-rating will be treated as a zero rating.

In the same way that we extended the collaborative filtering model to contain biases, we can do the same for latent factor based models. We can also introduce regularization in order to prevent our model from overfitting.

6.4 Other Ideas

In order to improve our models, we can also add in temporal biases and featuers. For example, if we want to put in a different time bias depending on when the data was collected, our model can change from $r_{xi} = \mu + b_x + b_i + q_i \cdot p_x$ to $r_{xi} = \mu + b_x(t) + b_i(t) + q_i \cdot p_x$

The best performing model for the Netflix challenge was an ensemble. If we have enough compute we can try models of models and put them all together in a complex way, and this approach can sometimes provide the best results.

7 Analysis of Large Graphs: Link Analysis

We can view the World Wide Web as a directed graph with nodes as webpages and edges as the hyperlinks that connect them. There are millions if not billions of webpages on the Web, but not all "web pages" have the same importance. We need to find a way to rank the pages in terms of importance, which we will do based off of the links of the webpages. This is called Link Analysis. We will cover 3 different approaches of Link Analysis for computing the importance of nodes in a graph: PageRank, Topic-Specific (Personalized) PageRank, and Web Spam Detection Algorithms.

7.1 PageRank

The idea behind PageRank is that the importance of a page is based off of its in-links (other pages that link to this page). However, not all in-links are equal. For example, if a webpage gets linked to by another "important" page, then this webpage is more important than if it was linked to by a not "important" page. Note that this is a recursive definition that we need to solve. The importance of a page can also be thought of like this: if we assume that people follow links randomly, this is equivalent to the limiting probability of someone being on that page.

For a given node j with n out-links, each of its out-links gets $\frac{r_j}{n}$ votes. Node j's importance is the sum of the votes of its in-links. Using this methodology, a "vote" from an important page is worth more, and a page is important if it is pointed to by other important pages. In mathematical terms, we define a rank r_j for page j to be $r_j = \sum_{i \to j} \frac{r_i}{d_i}$, where d_i is the out-degree of node i. To solve this equation, we add an additional constraint that the ranks of all the nodes must sum up to 1. Although we can solve this equation using Gaussian elimination from linear algebra, it is not computationally feasible when we have millions or billions of nodes. Hence, we instead use power iteration.

How we do this is we first define a our stochastic adjacency matrix M. If $i \to j$, then $M_{ji} = \frac{1}{d_i}$, else $M_{ji} = 0$. M is a column stochastic matrix as each column sums to 1. We then define our rank vector r, which is a column vector with one entry per page, where r_i is the importance score of page i, and $\sum_i r_i = 1$. Since we had defined before that $r_j = \sum_{i \to j} \frac{r_i}{d_i}$, we can rewrite this in matrix notation as $r = M \cdot r$

We observe that this means that r is an eigenvector of M and 1 is an eigenvalue of M. Therefore, by finding the principal eigenvector, we can solve this equation. We can solve this iteratively through power iteration. The steps would be as follows:

- Initialize: $r^{(0)} = [1/N, ..., 1/N]$
- Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
- Stop when $|r^{(t+1)} - r^{(t)}|_1 < \epsilon$

The above works assuming no dead ends of spider traps. A dead end is a node which has no out-links, so therefore the walker has no one to go. The importance will "leak out" and our power iteration method will converge to a vector of all 0's. A spider trap is a situation where all out-links are within a group, and the importance therefore only stays within this group and cannot escape. The power iteration method will not converge in this case and keep oscillating between moving the importance around the nodes in the spider trap.

The solution to spider traps is to use teleports. How this works is that at each time step, there is a probability β that we follow a random link, and a probability of $1 - \beta$ that we jump to a random node. If we set $\beta = 1$, we obtain what we previously had. Using teleports means that there is a certain probability at each time step that we can jump out of this spider trap. To solve the problem with the dead ends, we make it so that once we reach a dead end, we teleport with probability 1.

We now update our equation to the following to account for the teleports: $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta)\frac{1}{N}$. We assume no dead ends, but if we have dead ends, we can simply rescale r_j at each iteration to sum back to 1.

In matrix notation, we have $A = \beta M + (1 - \beta)[\frac{1}{N}]_{NXN}$, where $r = A \cdot r$. $[\frac{1}{N}]_{NXN}$ is an N by N matrix where all the entries equal $\frac{1}{N}$.

A is a dense matrix, and when we have large data, it can be difficult to store A, as well as r. Doing some math, we can simplify $r = A \cdot r$ to $r = \beta M \cdot r + [\frac{1-\beta}{N}]_{NXN}$. M is a sparse matrix, so now we can store it much easier.

One observation that makes this feasible even if r cannot be stored in main memory is that we can separate out different components of the vector r and the parts of M that correspond to those components and do each set of matrix multiplications separately (block by block).

An issue with PageRank are that it only looks at the generic popularity of a page, and is therefore biased against topic-specific authorities. This is where Topic-Specific PageRank comes in. In addition, it can be susceptible to link spam: artificial link topographies created in order to boost the page's rank.

7.2 Topic-Specific (Personalized) PageRank

The goal of topic-specific PageRank is to answer how important a page is based on a topic, rather than just ranking generically. We want to be able to evaluate Web pages not just according to their importance, but also by how close they are to a particular topic, e.g. "sports" or "history".

The key in topic-specific PageRank is that rather than allowing teleportation into any page at random, we can only teleport into a topic-specific set of "relevant" pages (teleport set). Formulaically, we have $A_{ij} = \beta M_{ij} + \frac{1-\beta}{S}; i \in S$ and $A_{ij} = \beta M_{ij} + 0$; otherwise.

One application of topic-specific PageRank is called SimRank, where we use this methodology to look at how similar two items are. How this works is that we make the SimRank teleport set equal only the starting point. For example, if we want to find how similar other nodes are to node u, we start a random walk from node u with a teleport set of only being able to teleport back to node u. In doing so, we can find out how similar other nodes are to node u.

7.3 TrustRank

Another application of topic-specific PageRank is used to detect Web spam. Once people had figured out that Google was using PageRank to rank the importance of pages, they spammed the system by creating many web pages with links that pointed to their own spam page in order to increase the PageRank.

What we find if you do the math is that if these "spam" pages all link to one another and no pages link to any page in this group, that the PageRank of all of these pages is 0. However, what the spammer can do is to go to a page, such as someone else's blog and post in the comment section a link to this group of spam pages. If a spammer does this from pages that DO have importance, what we find is that the PageRank of these spam pages do increase.

Therefore, to combat these spam farms, we can do a topic-specific PageRank with a teleport set of only trusted pages. Two ways of determining a set of trusted pages is to hard curate them using a real person, or to only use webpages that are .edu, or other trusted domains. Using these trusted pages, we can perform a topic-specific PageRank algorithm to determine the trust level of pages along the Web. The idea is that "good pages" link to other "good pages" and that it is rare for a "good page" to link to a "bad page." Therefore, by starting with trust pages and doing a random walk with a teleport set of only those trusted pages, we can have a trust score for each Web page, and ones with low scores can be determined as spam.

8 Community Detection in Graphs

The goal of community detection is to find groups of nodes, or clusters, within the large graph. For example in a graph of actors in movies, clusters can be formed with actors from the same country. What we are trying to do is find groups of nodes that are similar to each other and therefore in a group.

Sometimes the graph is extremely large, and our assumption is that although the graph fits in main memory, we cannot run any algorithm on it that is more than linear time. The algorithm we will develop will be proportional to the cluster size, even smaller than being proportional to the graph size!

The idea is that we can discover clusters based on a seed node (a starting node on the graph). Given this seed node s, we will do the random walk with a teleport set of only that seed node s. The idea is that if s belongs to a nice cluster, the random walk will get trapped within that cluster.

We need to define a metric to measure how good a cluster quality is. We want to maximize the number of within-cluster connections and minimize the number of between-cluster connections. We define a cut score $cut(A) = \sum_{i \in A, j \notin A} w_{ij}$. We are defining the score as the number of edges where one end is inside the cluster A and the other end is outside the cluster. We want to minimize the cut score, which solves the problem of minimizing the number of between-cluster connections. We then need to look at how to maximize the within-cluster connections.

To examine both between-cluster and within-cluster connections, we define our metric, called the conductance:

$\phi(A) = \frac{|\{(i,j) \in E; i \in A, j \notin A\}|}{min(vol(A), 2m - vol(A))}$, where $vol(A) = \sum_{i \in A} d_i$ (the total weight of the edges with at least one endpoint in A)

What this boils down to is that we take the cut score and then divide it by the number of edges in the minimum of A and A^C to make sure that we have lots of between-cluster connections as well.

Returning back to the algorithm, what we do is we run topic-specific page rank starting a node s with teleport set with only node s. We then sort the nodes by decreasing PageRank scores and calculate the conductance. We can plot a line chart with number of nodes on the x-axis and conductance of the y-axis. The local minima of this plot signal good clusters and tell us where to make the cuts.

We note that this process takes linear time. Every time we add a node, we can calculate the new cut score and volume by taking the previous one and adding/subtracting the impact of adding this new node to the cluster.

We can extend this concept to examine clusters of a certain motif. Currently, our "motif" is simply just an edge on an undirected graph. However, we can have different relationships on a directed graph and call that a "motif." All we need to do is to change our conductance metric to $\phi_M(S) = \frac{\#ofmotifscut}{vol_M(S)}$ and redraw the directed graph into a weighted undirected graph where each edge is weighted based on how many motifs it is adjacent to.

If we want to go beyond just clustering into groups, but rather want that as well as a hierarchical clustering (say something we could view in a dendogram), we can do that by defining a metric called modularity. Modularity Q is defined as the sum over all clusters of the difference between the number of edges and the "expected" number of edges. So how do we calculate the "expected" number of edges? The expected number of edges between node i and node j is $\frac{k_i k_j}{2m}$, where k_i is the degree of node i. This is the number of combinations of node i being able to connect to node j divided by the total number of edges. In other words, it is the expected probability that node i connects to node j.

Modularity Q is the then defined as:

$Q(G,S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} (A_{ij} - \frac{k_i k_j}{2m})$

Basically we are summing over all the clusters s and looking at the difference between the number of edges minus the expected number of edges for that cluster. We then divide by 2m to normalize between a value of -1 and 1.

In order to perform this hierachical clustering, we use the Louvain algorithm, where we can identify communities by maximizing the modularity. The Louvain algorithm is a greedy algorithm that runs in O(n log n) time. One step of the Louvain algorithm has 2 phases:

1. Modularity is optimized by allowing only local changes to node-communities memberships
2. The identified communities are aggregated into super-nodes to build a new network

We can then keep repeating these steps of the Louvain algorithm until we only have 1 total aggregate cluster.

For the first phase, we start by putting each node in a graph into a distinct community (one node per community). For each node i, we compute the modularity delta (ΔQ) when putting node i into the community of some neighbor j. We do this calculation for node i with all the different options of its neighbors j, and then we move i to the community j that yields the largest gain in ΔQ. Phase 1 runs until no movement yields a gain.

How we calculate ΔQ is that $\Delta Q = \Delta Q(i \to C) + \Delta Q(D \to i)$. What this is saying is that we combine the gain from putting i into the community C with the change in gain from taking i out of community D. Formulaically, we have:

$\Delta Q(i \to C) = [\frac{\sum_{in} + k_{i,in}}{2m} - (\frac{\sum_{tot} + k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2]$, where

- $\sum_{in}$ is the sum of link weights between nodes in C
- $\sum_{tot}$ is the sum of all link weights of nodes in C
- $k_{i,in}$ is the sum of link weights between node i and C
- k_i is the sum of all link weights (i.e., degree) of node i

Since all of the values in the formula are just summary statistics, the calculation can be done very quickly. In phase 2, the communities obtained in the first phase are contracted into super-nodes, and the network is created accordingly:

- Super-nodes are connected if there is at least one edge between the nodes of the corresponding communities
- The weight of the edge between the two supernodes is the sum of the weights from all edges between their corresponding communities

We then can repeat Phase 1 on the super-node network and continue repeating this algorithm until everything is combined into 1 node.

9 Graph Representation Learning

In order to run machine learning models on graphs to perform tasks such as link prediction or node classification, we need to find a way of representing each node in the graph as a vector. To do this, we map nodes to an embedding (similar to in NLP or Vision), but the problem can be quite complicated as there is no spatial locality or set structure.

Our goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network. In order to do this, we need to define an encoder, define a node similarity function, and optimize the parameters of the encoder so that the similarity in the embedding space approximates similarity in the original network. The key choice of methods will depend on how we define node similarity. Two examples of graph embeddings are DeepWalk and node2vec.

9.1 DeepWalk

With DeepWalk, we estimate the probability of visiting node v on a random walk starting from node u using some random walk strategy R. Then we optimize the embeddings to encode these random walk statistics. We define $N_R(u)$ as the neighborhood of u obtained by some strategy R. Let z_u be the embedding of node u. Then what we want to do is maximize the log-likelihood objective function: $max_z \sum_{u \in V} logP(N_R(u)|z_u)$.

To actually implement, we first run short fixed-length random walks starting from each node on the graph using some strategy R for each node u. For each node u, we collect the $N_R(u)$, the multiset of nodes visited on random walks starting from u. We can rewrite our objective function as a minimization problem. We want to minimize:

$L = \sum_{u \in V} \sum_{v \in N_R(u)} -log \frac{exp(z_u^T z_v)}{\sum_{n \in V} exp(z_u^T z_n)}$

However, we observe that to minimize the above takes quadratic time. We need to loop over every node u in V and then for every node n, calculate the dot product between z_u and z_n. We can use negative sampling in order to approximate the loss, and then we do not loop over ever node n each time to calculate the dot product. Essentially, we use the approximation that:

$log \frac{exp(z_u^T z_v)}{\sum_{n \in V} exp(z_u^T z_n)} \approx log(\sigma(z_u^T z_v)) - \sum_{i=1}^{k} log(\sigma(z_u^T z_{n_i}))$, where n_i are the selected nodes in the negative sample (usually around 5-20)

We then use our machine learning model to iteratively learn the embeddings using SGD to minimize the loss function. Then we have our embeddings! In order to generalize this concept/methodology, we can take a look at node2vec.

9.2 node2vec

The difference between DeepWalk and node2vec is that node2vec is more generalized. What we do in node2vec is relax the assumption on how we do the random walk. In DeepWalk, we run fixed-length, unbiased random walks starting from each node. In node2vec, we can bias the walks. While DeepWalk has a depth first search, in node2vec, we have two parameters p and q, which can determine if we want to do a depth first search walk or a breadth first search walk.

How this works is that at every step of the random walk, we can choose to return to our previous node, move to another node the same number of steps away from the starting node, or advance to a node that is farther away (in terms of the number of steps) from our starting node. We pick p and q such that the probability of returning to the previous node is $\frac{1}{p}$, the probability of moving to another node the same number of steps away is 1, and the probability of advancing to a farther away node is $\frac{1}{q}$. These are not normalized probabilities, so we normalize them so that they sum to 1. If we want a BFS type of walk, we select a low value of p. If we want a DFS type of walk, we select a low value of q. Then we run SGD on the same loss function as before and follow the steps of DeepWalk.

9.3 Embedding an Entire Graph

In addition to creating an embedding per node, we can also embed an entire graph into one vector. Two methodologies to get this embedding are: run a standard graph embedding technique on each node and then sum up the vectors to get the graph embedding, or introduce a "virtual node" to represent the graph and run a standard graph embedding technique on this virtual node.

10 Large-Scale Machine Learning

Large Scale machine learning is important because we have found that the size of the dataset matters more than the model itself when it comes to making good predictions. Google, in 2017, revisited this and published the Unreasonable Effectiveness of Data in Deep Learning, again finding that large data beats well-tuned models.

10.1 Decision Trees

A Decision Tree is a tree-structured plan of a set of attributes to test in order to predict the output. In a decision tree, we split the data at each internal node, and each leaf node makes a prediction. Although trees can split into more than two ways at each node, we will stick to only binary trees. In order to predict $\hat{y_i}$ given x_i, we simply drop x_i down the tree until it hits a leaf node and take that prediction.

When building a tree, we come across 3 questions:

1. How to split the tree at each node?
2. When to stop splitting?
3. How to predict?

10.1.1 How to split?

We have to decide at each node how to split the tree. We have many different attributes we can use at each split, so which one do we choose and at what value? The answer is we pick the attribute and value that optimizes some criterion, and this criterion will change depending on if we are doing a regression or a classification problem.

In regression, we will pick the attribute and value that maximizes the purity. Let D be the number of datapoints in the parent, D_L be the number of datapoints in the left child, and D_R be the number of datapoints in the right child. We find the split $(X^(i), v)$ that creates D, D_L, D_R such that we maximize the purity. Purity is defined as:

$|D| \cdot Var(D) - (|D_L| \cdot Var(D_L) + |D_R| \cdot Var(D_R))$, where $Var(D) = \frac{1}{|D|} \sum_{i \in D} (y_i - \bar{y})^2$

For classification problems, we split based on the attribute and value with the highest information gain. The information gain is defined as: $IG(Y|X) = H(Y) - H(Y|X)$, where H is the impurity measure, commonly selected as entropy, Gini index, or classification error. The intuition behind the information gain is what is the change in impurity if we did not do the split on this attribute, versus if we did. Entropy is defined as $H(X) = -\sum_{j=1}^{m} p(X_j) logp(X_j)$.

10.1.2 When to stop splitting?

Two heuristics for when we can stop splitting are when the leaf is "pure" (when the target variable $Var(y) < \epsilon$) or when the number of examples in a leaf is too small (|D| < 100).

If we keep splitting further and further, often times we can overfit, so that is one thing to be careful of.

10.1.3 How to predict?

There are many options for how to predict, but some ideas below for regression and classification include:

- Regression
  - Predict average y_i of the examples in the leaf
  - Build a linear regression model on the examples in the leaf
- Classification
  - Predict most common y_i of the examples in the leaf

10.1.4 Building Decision Trees Using MapReduce

Our assumptions are as follows:

- Tree is small (can keep it memory)
- Dataset too large to keep in memory
- Dataset too big to scan over on a single machine

We can use a learner called PLANET (Parallel Learner for Assembling Numerous Ensemble Trees) which takes a sequence of MapReduce jobs and builds a decision tree. With PLANET we assume that we are performing a regression task and that splits are binary. The tree will be built one level at a time with the following steps:

1. Master tells the Mappers which splits (n, $X^{(j)}$, v) to consider
2. MapReduce computes quality of those splits
3. Master then grows the tree for a level
4. Go back to Step 1

How the MapReduce works is that each Mapper gets a subset of data and computes partial statistics for a given split. The Reducers then collect partial statistics and output the final quality for a given split, so that the Master can make the final decision for where to split.

10.2 Support Vector Machines

With Support Vector Machines, we try to use a linear separator to separate out the data into two classes. The goal is to use the line that maximizes the margin between the closest data point for the positive and the negative training example. The line (or hyperplane in higher dimensions) should separate the positive and the negative examples.

Let w be a vector orthogonal to the separating hyperplane. Therefore, if we pick any vector u, then the dot product of w and u is the scaled projection of w onto u. Therefore, we can say that if $w \cdot u + b \geq 0$, then the example is positive, and if not then it is negative. In other words, $w \cdot u + b \geq 0$ is our decision boundary.

We then add constraints to our training data. We insist that $w \cdot x_+ + b \geq 1$, and $w \cdot x_- + b \leq -1$, where x_+ is a positive example and x_- is a negative example. We can rewrite this when we define y_i to be $+1$ if it is a positive example and -1 if it is a negative example. Now we have one equation: $y_i(w \cdot x_i + b) \geq 1$

To find out the width of the margin, we note that we get a vector $x_+ - x_-$ when we take the difference of the two vectors of the datapoints right on the support. We take $x_+ - x_-$ and project it onto w and divide by the magnitude of w to make it a unit vector. Plugging in the formula for x_+ and x_- from before, we calculate the width of the margin to be $\frac{2}{||w||}$. Our goal is maximize this quantity, which is equivalent to minimizing $\frac{1}{2}||w||^2$, subject to the constraint that $y_i(w \cdot x_i + b) \geq 1$.

If the data is not linearly separable, then we introduce a penalty term that consists of "slack variables." The slack variable penalizes the datapoint by taking the value of 0 if it is classified properly, and the distance of the error if it is classified improperly. Mathematically, we are now minimizing $\frac{1}{2}||w||^2 + C\sum_{i=1}^{n} max\{0, 1 - y_i(w \cdot x_i + b)\}$, where C is the regularization hyperparamter set by the user.

To compute the minimizer, which is a quadratic term, we can use a solver. In practice, the solver would use gradient descent or stochastic gradient descent in order to compute the optimal solution.

11 Mining Data Streams

In many data mining situations, we do not know the entire data set in advance. We can think of the data as infinite and non-stationary (the distribution changes over time) as the stream comes in. The system cannot store the entire stream accessibly, so we have to come up with interesting algorithms. Some examples of questions that one wants to answer on a data stream are how to sample data from a stream and how to perform a query over sliding windows.

11.1 Sampling from a Data Stream

Since we can not store the entire stream, we need to instead store a sample. The naive way of storing a sample would be to store a fixed proportion, say $\frac{1}{10}$th of the elements in the stream. However, we can run into problems with answering certain questions with this approach. For example, if we want to answer: What fraction of unique queries by an average search engine user are duplicates, we will have a problem. The reason if because if we took 1 out of 10 queries into our sample, the chances of having a duplicate are smaller than if we took the whole set of queries, since for the sample to have a duplicate, that query must be randomly chosen from the whole set of queries twice. The solution to this problem is to pick $\frac{1}{10}$th of users and take all their searches into the sample.

The better way to take sample a stream is to maintain a fixed size sample, even as the stream grows. We can do this using a technique called reservoir sampling. Suppose we want a sample of size s. How we do reservoir sampling is that we take the first s elements of the stream and store it into the sample. As each element after the first s arrives one at a time (nth element with n > s), with probability $\frac{s}{n}$ we keep the nth element, otherwise we discard it. If we keep the nth element in the sample, then uniformly we discard of the existing s elements currently in the sample. What we can prove is that after n elements, the sample contains each element seen so far with probability $\frac{s}{n}$.

11.2 Queries Over a Long Sliding Window

Practically, we may want to examine data over a sliding window of size N, but N can be so large that it cannot fit into memory or even disk. For example, say at Amazon, we want to find how many times we have sold product X in the last N sales. For each product, we represent a binary string of 0's and 1's for each transaction at Amazon, 0 if the product was not bought in that transaction, and 1 if it was.

Therefore, we generalize the problem to be that given a stream of 0's and 1's, how many 1's are in the most recent N bits? Since we cannot store the sliding window N into memory or in disk, we cannot get an exact answer, so we will approximate it. If we assume a uniform distribution of 0's and 1's (which is unrealistic), we can just take the proportion of history 0's and 1's and multiply it by the size of the window N. However, in real life this is not practical as most streams are not uniform and the distribution can change over time, so we need another method.

The DGIM method is a solution that does not assume uniformity. The solution gives an approximate answer, but it never off by more than 50%. By "off by 50%", we are referring to the difference in the approximate number of 1's to the true number of 1's.

How DGIM works is that we summarize blocks with a specific number of 1's, and as the blocks are further back, they increase exponentially in the number of 1's they contain. We call these buckets. The number of 1's per bucket are increasing in a power of 2. For example, the most recent buckets will have 1 1 in them, the following buckets with have 2 1's in them, the ones after that will have 4 1's in them, etc. Each bucket is distinct and nonoverlapping, so we also store the end time stamp of each bucket to figure out where each bucket starts and ends. As new elements are added to the stream, the sliding window will include these new elements and "forget" old ones. If a new element added is a 0, the buckets will remain the same. If a new element added is a 1, then the buckets will need to readjust to account for this 1. For this algorithm, we pick a value of r, which is how many buckets maximum can contain this certain value of 1's.

For example if r = 2, then we can have at most 2 buckets at one time containing 1 1, or 2 1's, or 4 1's etc. If when a new element is added, we end up creating 3 buckets containing 1 1, then we merge the oldest 1 into a bucket of 2, and this will cascade until we have at most 2 buckets of each amount of 1's. A bucket remains in memory, until all elements of that bucket have moved out of the sliding window N. If part of the bucket still remains in memory, we still include that bucket in our calculations.

To estimate the number of 1s in the most recent N bits, we first sum the sizes of all buckets, but the last. Then we add half the size of the last bucket to this sum, and we get our approximate answer. It can be proven that the error of this approximate answer is at most 50% for r = 2. If we want more accuracy, we can increase r (but will have to store more bits), and we can show that the error is at most $O(\frac{1}{r})$.

This methodology can be applied for any k, where k $\leq$ N. So we can find out how many 1's are in the most recent k bits, when k is inside the sliding window of N. If we have a stream of positive integers (instead of 0 and 1 only), we can also extend this concept to take the sum of the last k elements. If we know that all integers will have less than m bits representing them, we take a stream for each integer for each bit value and perform the same methodology. For example, if our integer is at most 2^32, we can have 32 sliding windows and perform our count of 1's in each sliding window, and put it back together into base 10 to find out our approximate sum.

11.3 Filtering Data Streams

The goal is that given data that is a tuple in a stream, we want to filter out tuples that have a key in our list of keys that we want, S.

What we do is given a set of keys S that we want to filter, we create a bit array B of length larger than S, starting with of 0's. We then hash each member $s \in$ S to one of n buckets, and set that bit representing that bucket from a 0 to a 1. We can then hash each element a of the stream and output only those that hash to bit that was set to 1. This creates some false positives, but no false negatives. What we realize in this case is that our probability of a false positive is:

$1 - (1 - \frac{1}{n})^m = 1 - (1 - e^{\frac{-m}{n}})$, where m is the length of S (number of keys), and n is the length of B.

An improvement to this is to use a Bloom filter. Instead of using 1 hash function, we use k independent hash functions, and hash each element $s \in$ S into B using each hash function. When a stream element a arrives, we return it only if for every single hash function k, it hashes into a bucket with a 1. Again, we have no false negatives, but now our probability of a false positive is:

$1 - (1 - e^{\frac{-km}{n}})^k$

As we increase k, we decrease our probability of a false positive, up until an optimal point. Past this point, increasing k will increase our probability of a false positive. We choose the optimal k. One small thing to note is that it is actually mathematically equivalent to hash all $s \in$ S into one large B, or to k small B's, but since keeping one large B is simpler, that is common practice.

11.4 Counting Distinct Elements

Often, we want to maintain a count of the number of distinct elements seen so far in a data stream. For example, a business may want to know how many distinct products they have sold in the last week. We assume that we do not have space to maintain the set of elements seen so far, yet we want to estimate the count in an unbiased way.

We can do this using the Flajolet-Martin approach. How this works is that we pick a hash function h that maps each of the N elements to at least $log_2 N$ bits. For each stream element a, let $r(a)$ be the number of trailing 0s in $h(a)$. For example, if $h(a) = 12$, then 12 is 1100 in binary, so $r(a) = 2$. We record R, which is the the maximum $r(a)$ seen in the stream thus far. Our estimate of the number of distinct elements is 2^R.

The rough intuition as to why this approach works is as follows. Since $h(a)$ hashes a with equal probability to any of N values, then $h(a)$ is a sequence of $log_2 N$ bits, where 2^{-r} fraction of all a's have a tail of r zeros. For example, about 50% of a's hash to ***0, 25% hash to **00 etc. So if the longest tail we saw was R = 2, then we expect to have seen about 4 distinct items so far.

The issue with this methodology is the $E[2^R]$ is infinite, therefore as the stream grows, so will the estimate. The workaround is to use many hash functions h_i and get many samples of R_i. We can calculate each of these sample R_i's by taking the median in each sample as R_i, rather than the max. Then we take all the R_i's from all the groups and calculate the average. This is our final R, and then 2^R gives us our estimate of the distinct elements.

11.5 Computing Moments

If we let m_i be the number of times value i occurs in the stream, then the kth moment is defined as $\sum_{i \in A} (m_i)^k$. Note that this is a power of the count of the occurrence of i, not the value of i itself. Some interesting moments we may want to calculate are the 0th moment, which tells us the number of distinct elements, the 1st moment, which tells us the total number of elements, and the 2nd moment, which tells us how uneven the distribution is.

To do this on a stream, we can use the AMS method. How the AMS method works is that we uniformly pick some random time t where t $<$ n. For that element at time t, we count the number of times that item occurs from time t to time n. We store the tuple (item value, count). We estimate the $f(X_j)$ for item j as $f(X_j) = n(c^k - (c-1)^k)$. We then calculate our

moment S as $S = \frac{1}{n}\sum_{j=1}^{n} f(X_j)$. We can prove that in expectation this equals the kth moment.

In practice, what we would do is compute $f(X_j)$ for as many variables X_j as we could fit in memory. We would divide them into groups and calculate the average for each group, and then take the median of the averages as our final estimate.

One problem with this approach is that streams never end, so the value of n is always increasing. How we can solve this is to use reservoir sampling. We include the first m times in our sample for the first m variables in the stream. When the nth element arrives (n > m), choose it with probability $\frac{m}{n}$. If we choose to include it in the sample, then we uniformly throw out one of the previously stored variables X_j in our sample with equal probability.

11.6 Counting Itemsets

We want to count itemsets in a stream. Since there are many combinations of different items, it is too to expensive to store a stream for each itemset and perform DGIM on it. To solve this problem, we use exponentially decaying windows.

What we will answer is the question of: what are "currently" the most frequent itemsets? We keep a current count for each itemset. How exponentially decaying windows works is we take the answer at time t to be: $Count_j = \sum_{i=1}^{t} a_i(1-c)^{t-i}$, where c is a small constant such as 10^{-6} or 10^{-9}.

Computationally, how this would work is that each time a new element comes into the stream, we multiply the current count for that item by $1 - c$, and then add 1 to it if the new element is our itemset of interest.

In practice, there may be too many itemsets to keep a count of each different itemset count in memory, so we will only keep "important" ones. What we can do is create a weight, such $\frac{1}{2}$, and if any of our counts drop below that weight, then we discard it in memory (because it has decayed a lot, meaning it is in the past and not recent).

12 Advertising on the Web

We want to match advertisers with opportunities to show an ad. This is an online problem, since we have to make decisions as queries/topics show up. We do not know what topics will show up in the future. As opposed to offline algorithms where we have all the data in advance, in online, we only see the data as it comes in (similar to streams).

In a bipartite graph where we want to match boys with girls, we first examine the performance of a greedy algorithm. The greedy algorithm takes a girl one at a time, and pairs the new girl with any eligible boy (connected to the girl on the graph). To find the performance of the greedy algorithm, we first must define the competitive ratio. The competitive ratio = the minimum over all possible inputs I of the cardinality of M greedy divided by the cardinality of M optimal (the best possible matching if we knew the whole graph in advance). Basically, we are comparing what the worst case scenario for the greedy algorithm is divided by the best case optimal solution. We can prove that the competitive ratio for the greedy algorithm is 0.5. This means in the worst case scenario, using the greedy algorithm, we match half the amount of the boys/girls that we would have matched if we knew the entire graph beforehand and calculated the optimal pairings.

Relating this to web advertising, we want to match the advertisers with the opportunities to show an ad. When web advertising began, advertisers were changed per impression of the the ad, which means they were charged each time their ad was shown. Overture and Adwords changed the model so that the advertisers would only pay when their shown ad was clicked on. Therefore, the goal for search engines now is to maximize their revenue by maximizing the product of the click-through-rate and the bid of the advertiser.

In practice, the challenge is that each advertiser has a limited budget to spend, as well as the CTR (click-through-rate) is unknown. In practice, CTR is usually calculated by predicting based on past history.

Assuming we know the advertiser's limited budget and their CTR, how can we maximize the revenue for the search engine? More specifically, the Adwords problem is the following:

- Given:
  1. A set of bids by advertisers for search queries
  2. A click-through rate for each advertiser-query pair
  3. A budget for each advertiser (say for 1 month)
  4. A limit on the number of ads to be displayed with each search query
- Respond to each search query with a set of advertisers such that:
  1. The size of the set is no larger than the limit on the number of ads per query
  2. Each advertiser has bid on the search query
  3. Each advertiser has enough budget left to pay for the ad if it is clicked upon

We can use the BALANCE algorithm to try to maximize the revenue for the search engine given that we are working online, and do not know all the queries in advance, only as they come in one by one. Assuming the bids are all equal as well as the budgets for each advertiser, the BALANCE algorithm is simple: for each query, pick the advertiser with the largest unspent budget. We note that using this strategy, BALANCE must exhaust at least one budget of an advertiser. We can prove that the worst competitive ratio of BALANCE is $1 - \frac{1}{e} \approx 0.63$.

We relax the constraint that all bids and budgets must be equal. We then define the following for each query q, bidder i, bid x_i, budget b_i, amount spent so far m_i, fraction of budget left over $f_i = 1 - \frac{m_i}{b_i}$. We then define $\psi_i(q) = x_i(1 - e^{-f_i})$. Using BALANCE, we query q to bidder i with largest value of $\psi_i(q)$. We can prove that the worst competitive ratio in this general case remains $1 - \frac{1}{e} \approx 0.63$.

13 Learning Through Experimentation

For web advertising, as well as many other application where we want to maximize reward while investigating different options, we need to solve a multi-armed bandit problem. In a k-armed bandit problem, we have a different arms, or options, each with a reward of 1 with probability μ_a and a reward of 0 with probability $1 - \mu_a$. We also assume each draw is independent, and the question is how to maximize the total reward? We note that on the first draw, we know nothing about the probability of winning with each arm, so we want to get a good idea of the distribution of winning for each arm, yet at the same time want to maximize reward while performing this investigation.

Our performance metric is called regret. Let μ_a be the reward for arm a, and the payoff/reward for the best arm be $\mu^* = max_a \mu_a$. Let $i_1, i_2, ..., i_T$ be the sequence of arms pulled. The instantaneous regret at time t is defined as $r_t = \mu^* - \mu_{i_t}$, and the total regret is defined as $R_T = \sum_{t=1}^{T} r_t$. We want a methodology that guarantees that $\frac{R_T}{T} \to 0$ as $T \to \infty$.

If we knew all the payoffs, we would just pull the arm with the highest payoff all the time. If we only cared about estimating the payoffs, we would pick each of the k arms at random. This is called the trade off between exploitation and exploration.

So how to do this? We note that using a greedy algorithm will not work. Take the example where arm A_1 has a reward with probability 0.3 and A_2 has a reward with probability 0.7. If we get unlucky and pull A_1, which happens to get the reward, and then pull A_2, which happens not to on one try, then if we use the greedy algorithm we will only pull A_1 from here on out. Therefore, we can see that using a greedy algorithm can cause us to converge to a suboptimal solution, since it does not perform exploration sufficiently.

Instead, we can use an epsilon-greedy algorithm. How this works is that we set a probability ϵ_t. With probability ϵ_t we explore an arm by picking one uniformly at random, and with probability $1 - \epsilon_t$ we exploit by picking the arm with the highest empirical mean payoff. We can prove that the epsilon-greedy algorithm satisfies the following: $\frac{R_T}{T} \to 0$ as $T \to \infty$. However, some issues with the epsilon-greedy method include that we concretely fix an ϵ, and also that in the exploration we select arms uniformly at random (so each arm is equally likely to be picked), and we know that some of these arms can be suboptimal.

A key insight is that when we explore, we want to compare all the arms to select which arm to explore next. We can do this not just by looking at the mean, but rather the confidence. For example, an arm that we have explored many times will give us a closer confidence interval for what μ_a of that arm is that an arm we have only explored once.

So how do we calculate confidence bounds for each arm? We can use Hoeffding's Inequality. This inequality provide an upper bound on the probability that the average deviates from its expected value by more than a certain amount. This inequality states that to find out the confidence interval b (for a given confidence level δ), we have the following formula:

$b \geq \sqrt{\frac{ln(\frac{2}{\delta})}{2m}}$, where m is the number of observations

Using this equation, we come up with a new technique called the UCB1 (Upper Confidence Sampling) algorithm. How this works is we start by setting $\hat{\mu_1} = \hat{\mu_2} = ... = \hat{\mu_k} = 0$ and $m_1 = m_2 = ... = m_k = 0$. For each arm, we calculate $UCB(a) = \hat{\mu_a} + \alpha\sqrt{\frac{2lnt}{m_a}}$. We take the arm with the largest $UCB(a)$ and pull it. For this arm j, we then update $m_j \leftarrow m_j + 1$ and $\hat{\mu_j} \leftarrow \frac{1}{m_j}(y_t + (m_j - 1)\hat{\mu_j})$. α is our free parameter used for trading off between exploration and exploitation.

The result of this algorithm is that it ensures each arm is tried infinitely often but still balances exploration and exploitation. This is because the arms tried less often will have a larger upper bound on their confidence interval, and then they will then be tried. Once they are tried more often, their confidence interval will shrink, and then another arm tried less often will then be tried, etc.

14 Optimizing Submodular Functions

When we recommend content to a user, we want to avoid recommending similar items over and over again to that user. For example, when showing users trending news, we don't want to recommend 10 articles to the user that all are covering the same topic. Therefore, we want diversity in our recommendations and we want to optimize for that diversity.

Assume in the news coverage example that we are allowed to show the user k documents. Each document i covers a set X_i of concepts/words/ideas. Those k documents are in the set A, which is a subset of all the documents. We define $F(A) = |\cup_{i \in A} X_i|$. We want to maximize $F(A)$. In other words, we want to maximize how many different concepts are altogether in those k documents.

To find this maximum is actually an NP-complete problem, making it computationally expensive. However, instead, we can use a greedy algorithm, one that adds an extra document one at a time, and at each step it adds the document that maximizes the $F(A)$ at that time.

We can prove that this greedy algorithm produces a solution A where $F(A) \geq (1 - \frac{1}{e}) * F(A)_{opt}$. Basically, we can show that the greedy algorithm performs no worse than 63% of the coverage of the optimal solution. This claim only holds for functions that have the following 2 properties:

- F is Monotone
    1. Adding more documents does not decrease coverage
    2. If $A \subseteq B$, then $F(A) \leq F(B)$ and $F(\{\}) = 0$
- F is Submodular
    1. Adding an element to a set gives less improvement than adding it to one of its subsets

14.1 Submodularity

Two equivalent definitions of submodularity:

1. A set function $F(\cdot)$ is called submodular if for all $A, B \subseteq W$: $F(A) + F(B) \geq F(A \cup B) + F(A \cap B)$
2. A set function $F(\cdot)$ is called submodular if for all $A \subseteq B$: $F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$
    - What this is saying is that the gain of adding d into a small set is larger than the gain of adding d into a larger set (a superset)
    - This is the diminishing returns characterization of submodularity

One important property of submodularity is that it is closed under non-negative linear combinations. Hence, if $F_1 ... F_m$ are submodular and $\lambda_1 ... \lambda_m > 0$, then $F(A) = \sum_{i=1}^{m} \lambda_i F_i(A)$. Therefore, if we take a weighted average of submodular functions, our resulting function is still submodular.

14.2 Probabilistic Set Cover

Earlier, we had proved that by using the greedy algorithm, we can cover at least 63% of the concepts that we would have covered in the optimal case. However, this assumes equal weighting of each concept, when realistically, a user might care about certain concepts more than others. Each concept then is provided a weight w_c.

In addition, we define $cover_d(c)$ the probability that the document d covers concept c. Therefore, the probability that at least one document in set A contains the concept c is $cover_A(c) = 1 - \prod_{d \in A}(1 - cover_d(c))$. Our objective function now that we want to maximize is $max_{A:|A| \leq k} F(A) = \sum_c w_c cover_A(c)$. Using the property that non-negative linear combinations of submodular functions are also submodular, this new objective function is also submodular. Therefore, the greedy algorithm here also performs (in terms of coverage) at least 63% as well as the optimal solution.

14.3 Lazy Evaluation of Submodular Functions

In practice, if we were to run the greedy algorithm naively, it would be quadratic and therefore slow. Each time, we would have to re-evaluate the marginal gains of adding each document and see which one gives us the best marginal gain to include that in the set. We can greatly speed this up by using lazy evaluation.

By the submodularity definition of diminishing returns, we know that a set function $F(\cdot)$ is called submodular if for all $A \subseteq B$: $F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$. If we define A_i to be the set we choose using the greedy algorithm at time i and A_j to be the set we choose at time j, where j is at a later time step than i, what we realize is that $A_i \subseteq A_j$. Therefore, we observe that as time passes and we add more elements to A, the marginal increase that each element can add to A individually shrinks.

Using this observation, we can implement lazy evaluation. How this works is that we keep an ordered list of marginal benefits Δ_i from previous iterations. We then re-evaluate Δ_i only for the top element, and then resort. For example, in the first pass, we look at the marginal benefit of each element and then sort them from greatest to least, and add the largest one to the set. On the second pass, we start by looking at the largest element remaining from the previous step and calculate the new marginal benefit. If it is still larger than the one right below it, then we can add it to the set. If it is now less than the one right below it, we can move it to the sorted position where it now belongs (sorted by marginal benefit).

14.4 Determining the Concept Weights

We can start with equal weights for all the concepts, normalized so that the sum is 1. When we show a recommendation to the user, if the user engages with a document that has that concept, we can increase the value of the weight, and if the user does not engage with documents with that concept, we can decrease the value of the weight.

One way of doing this is to to define $r = +/-1$, depending on whether or not a user engaged with a document with that concept. We can set a value for $\beta > 1$ as a scaling factor. For each concept $c \in X_d$, we set $w_c = \beta^r w_c$. Therefore, if r is $+1$, then the weight will increase by a factor of β and if r is -1, then it will decrase by a factor of β. We then normalize the weights to sum up to 1 again.