MapReduce and Frequent Itemsets Mining

Yang Wang
MapReduce (Hadoop)

Programming model designed for:

- Large Datasets (HDFS)
  - Large files broken into chunks
  - Chunks are replicated on different nodes

- Easy Parallelization
  - Takes care of scheduling

- Fault Tolerance
  - Monitors and re-executes failed tasks
MapReduce

3 Steps

- **Map:**
  - Apply a user written map function to each input element.
  - The output of Map function is a set of key-value pairs.

- **GroupByKey:**
  - Sort and Shuffle: Sort all key-value pairs by key and output key-(list of value pairs)

- **Reduce**
  - User written reduce function applied to each key-[list of value] pairs
Map-Reduce: A diagram

MAP:
Read input and produces a set of key-value pairs

Intermediate

k1:v k1:v k2:v
k1:v k3:v k4:v
k4:v k5:v
k1:v k3:v

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Grouped

k1:v k2:v k3:v
k4:v k5:v

Reduce:
Collect all values belonging to the key and output

Output
Coping with Failure

MapReduce is designed to deal with compute nodes failing.

Output from previous phases is stored. Re-execute failed tasks, not whole jobs.

**Blocking Property**: no output is used until the task is complete. Thus, we can restart a Map task that failed without fear that a Reduce task has already used some output of the failed Map task.
Data Flow Systems

- **MapReduce uses two ranks of tasks:**
  - One is Map and other is Reduce
  - Data flows from first rank to second rank
- **Data Flow Systems generalise this:**
  - Allow any number of tasks
  - Allow functions other than Map and Reduce
- **Spark is the most popular data-flow system.**
  - RDD’s : Collection of records
  - Spread across clusters and read-only.
Frequent Itemsets

- The Market-Basket Model
  - Items
  - Baskets
  - Count how many baskets contain an itemset
  - Support threshold => frequent itemsets

- Application
  - Confidence
    - \( \Pr(D \mid A, B, C) \)
Computation Model

- Count frequent pairs
- Main memory is the bottleneck
- How to store pair counts?
  - Triangular matrix/Table
- Frequent pairs -> frequent items
- A-Priori Algorithm
  - Pass 1 - Item counts
  - Pass 2 - Frequent items + pair counts
- PCY
  - Pass 1 - Hash pairs into buckets
    - Infrequent bucket -> infrequent pairs
  - Pass 2 - Bitmap for buckets
    - Count pairs w/ frequent items and frequent bucket
Main-Memory: Picture of A-Priori

Pass 1: Item counts
Pass 2: Frequent items
Counts of pairs of frequent items (candidate pairs)

Green box represents the amount of available main memory. Smaller boxes represent how the memory is used.
Main-Memory: Picture of PCY

Main memory

Pass 1
- Item counts
- Hash table for pairs

Pass 2
- Frequent items
- Bitmap
- Counts of candidate pairs
All (Or Most) Frequent Itemsets

- Handle Large Datasets
- Simple Algorithm
  - Sample from all baskets
  - Run A-Priori/PCY in main memory with lower threshold
  - No guarantee
- SON Algorithm
  - Partition baskets into subsets
  - Frequent in the whole => frequent in at least one subset
- Toivonen’s Algorithm
  - Negative Border - not frequent in the sample but all immediate subsets are
  - Pass 2 - Count frequent itemsets and sets in their negative border
  - What guarantee?
Locality Sensitive Hashing and Clustering

Hongtao Sun
Locality-Sensitive Hashing

Main idea:

- **What**: hashing techniques to map similar items to the same bucket → candidate pairs
- **Benefits**: $O(N)$ instead of $O(N^2)$: avoid comparing all pairs of items
  - **Downside**: false negatives and false positives
- **Applications**: similar documents, collaborative filtering, etc.

For the similar document application, the main steps are:

1. **Shingling** - converting documents to set representations
2. **Minhashing** - converting sets to short signatures using random permutations
3. **Locality-sensitive hashing** - applying the “b bands of r rows” technique on the signature matrix to an “s-shaped” curve
Locality-Sensitive Hashing

Shingling:

• Convert documents to set representation using sequences of $k$ tokens
• Example: abcabc with $k = 2$ shingle size and character tokens $\rightarrow \{ab, bc, ca\}$
• Choose large enough $k \rightarrow$ lower probability shingle $s$ appears in document
• Similar documents $\rightarrow$ similar shingles (higher Jaccard similarity)

Jaccard Similarity: $J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$

Minhashing:

• Creates summary signatures: short integer vectors that represent the sets and reflect their similarity
Locality-Sensitive Hashing

General Theory:

● Distance measures $d$ (similar items are “close”):
  ○ Ex) Euclidean, Jaccard, Cosine, Edit, Hamming

● LSH families:
  ○ A family of hash functions $H$ is $(d_1, d_2, p_1, p_2)$-sensitive if for any $x$ and $y$:
    ■ If $d(x, y) \leq d_1$, $\Pr[h(x) = h(y)] \geq p_1$; and
    ■ If $d(x, y) \geq d_2$, $\Pr[h(x) = h(y)] \leq p_2$.

● Amplification of an LSH families ("bands" technique):
  ○ AND construction ("rows in a band")
  ○ OR construction ("many bands")
  ○ AND-OR/OR-AND compositions
Locality-Sensitive Hashing

Suppose that two documents have Jaccard similarity $s$.

Step-by-step analysis of the banding technique ($b$ bands of $r$ rows each)

- Probability that signatures agree in all rows of a particular band:
  - $s^r$
- Probability that signatures disagree in at least one row of a particular band:
  - $1 - s^r$
- Probability that signatures disagree in at least one row of all of the bands:
  - $(1 - s^r)^b$
- Probability that signatures agree in all rows of a particular band
  $\Rightarrow$ Become candidate pair:
  - $1 - (1 - s^r)^b$
Locality-Sensitive Hashing

A general strategy for **composing families** of minhash functions:

**AND construction (over \( r \) rows in a single band):**

- \((d_1, d_2, p_1, p_2)\)-sensitive family ⇒ \((d_1, d_2, p_1^r, p_2^r)\)-sensitive family
- Lowers all probabilities

**OR construction (over \( b \) bands):**

- \((d_1, d_2, p_1, p_2)\)-sensitive family ⇒ \((d_1, d_2, 1 - (1 - p_1)^b, 1 - (1 - p_2)^b)\)-sensitive family
- Makes all probabilities rise

We can try to make \( p_1 \to 1 \) (lower false negatives) and \( p_2 \to 0 \) (lower false positives), but this can require many hash functions.
Clustering

What: Given a set of points and a distance measure, group them into "clusters" so that a point is more similar to other points within the cluster compared to points in other clusters (unsupervised learning - without labels)

How: Two types of approaches

- **Point assignments**
  - Initialize centroids
  - Assign points to clusters, iteratively refine

- **Hierarchical:**
  - Each point starts in its own cluster
  - Agglomerative: repeatedly combine nearest clusters
Point Assignment Clustering Approaches

- Best for spherical/convex cluster shapes
- **k-means**: initialize cluster centroids, assign points to the nearest centroid, iteratively refine estimates of the centroids until convergence
  - Euclidean space
  - Sensitive to initialization (K-means++)
  - Good values of “k” empirically derived
  - Assumes dataset can fit in memory
- **BFR algorithm**: variant of k-means for very large datasets (residing on disk)
  - Keep running statistics of previous memory loads
  - Compute centroid, assign points to clusters in a second pass
Hierarchical Clustering

- Can produce clusters with unusual shapes:
  - e.g. concentric ring-shaped clusters
- **Agglomerative approach:**
  - Start with each point in its own cluster
  - Successively merge two “nearest” clusters until convergence
- **Differences from Point Assignment:**
  - Location of clusters: centroid in Euclidean spaces, “clustroid” in non-Euclidean spaces
  - Different intercluster distance measures: e.g. merge clusters with smallest $\textit{max}$ distance (worst case), $\textit{min}$ distance (best case), or $\textit{average}$ distance (average case) between points from each cluster
  - Which method works best depends on cluster shapes, often trial and error
Dimensionality Reduction and Recommender Systems

Jayadev Bhaskaran
Dimensionality Reduction

- Motivation
  - Discover hidden structure
  - Concise description
    - Save storage
    - Faster processing

- Methods
  - SVD
    - $M = U \Sigma V^T$
      - $U$ user-to-concept matrix
      - $V$ movie-to-concept matrix
      - $\Sigma$ “strength” of each concept
  - CUR Decomposition
    - $M = CUR$
SVD

- \( M = U \Sigma V^T \)
  - \( U^T U = I, V^T V = I, \Sigma \) diagonal with non-negative entries
  - Best low-rank approximation (singular value thresholding)
  - Always exists for any real matrix \( M \)

- Algorithm
  - Find \( \Sigma, V \)
    - Find eigenpairs of \( M^T M \) -> \((D, V)\)
    - \( \Sigma \) is square root of eigenvalues \( D \)
    - \( V \) is the right singular vectors
    - Similarly \( U \) can be read off from eigenvectors of \( MM^T \)
  - Power method: random init + repeated matrix-vector multiply (normalized) gives principal eigenvector
  - Note: Symmetric matrices
    - \( M^T M \) and \( MM^T \) are both real, symmetric matrices
    - Real symmetric matrix: eigendecomposition \( Q \Lambda Q^T \)
CUR

- M = CUR
- Non-uniform sampling
  - Row/Column importance proportional to norm
  - U: pseudoinverse of submatrix with sampled rows R & columns C
- Compared to SVD
  - Interpretable (actual columns & rows)
  - Sparsity preserved (U,V dense but C,R sparse)
  - May output redundant features

\[
\text{SVD: } A = U \Sigma V^T
\]

\[
\text{CUR: } A = C U R
\]
Recommender Systems: Content-Based

**What:** Given a bunch of users, items and ratings, want to predict missing ratings

**How:** Recommend items to customer x similar to previous items rated highly by x

- **Content-Based**
  - Collect user profile \(x\) and item profile \(i\)
  - Estimate utility: \(u(x,i) = \cos(x,i)\)
Recommender Systems: Collaborative Filtering

- **user-user CF vs item-item CF**
  - user-user CF: estimate a user’s rating based on ratings of similar users who have rated the item; similar definition for item-item CF

- **Similarity metrics**
  - Jaccard similarity: *binary*
  - Cosine similarity: treats missing ratings as “negative”
  - Pearson correlation coeff: remove mean of non-missing ratings (standardized)

- **Prediction of item i from user x:** \( s_{xy} = \text{sim}(x,y) \)
  \[
  r_{xi} = \frac{\sum_{y \in N} s_{xy} r_{yi}}{\sum_{y \in N} s_{xy}}
  \]

- Remove baseline estimate and only model rating deviations from baseline estimate, so that we’re not affected by user/item bias
Recommender Systems: Latent Factor Models

Motivation: Collaborative filtering is a local approach to predict ratings based on finding neighbors. Matrix factorization takes a more global view.

Intuition: Map users and movies to (lower-dimensional) latent-factor space. Make prediction based on these latent factors.

Model: $\hat{r}_{xi} = p_x \cdot q_i$ for user x and movie i
Recommender Systems: Latent Factor Models

\[ \min_{P,Q} \sum_{(x,i) \in \text{training}} (r_{xi} - p_x \cdot q_i)^2 + \lambda_1 \sum_x \|p_x\|^2 + \lambda_2 \sum_i \|q_i\|^2 \]

- Only sum over observed ratings in the training set
- Use regularization to prevent overfitting
- Can solve via SGD (alternating update for P, Q)
- Can be extended to include biases (and temporal biases)

\[ \hat{r}_{xi} = \mu + b_x + b_i + p_x \cdot q_i \]
PageRank

Lantao Mei
PageRank

- PageRank is a method for determining the importance of webpages
  - Named after Larry Page
- Rank of a page depends on how many pages link to it
- Pages with higher rank get more of a vote
- The vote of a page is evenly divided among all pages that it links to
Example

- \( r_a = \frac{r_y}{2} + r_m \)
- \( r_y = \frac{r_y}{2} + \frac{r_a}{2} \)
- \( r_m = \frac{r_a}{2} \)
Example

Deal with pathological situations by adding a random teleportation term

• \( r_a = 0.8 \left( \frac{r_y}{2} + r_m \right) + 0.2/3 \)
• \( r_y = 0.8 \left( \frac{r_y}{2} + \frac{r_a}{2} \right) + 0.2/3 \)
• \( r_m = 0.8 \left( \frac{r_a}{2} \right) + 0.2/3 \)
PageRank

If $i \to j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$

- **PageRank equation** [Brin-Page, ‘98]
  \[
  r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}
  \]

- **The Google Matrix $A$:**
  \[
  A = \beta M + (1 - \beta) \left[ \frac{1}{N} \right]_{N \times N}
  \]

- We have a recursive problem: $r = A \cdot r$
Topic-specific PageRank

- Teleport can only go to a topic-specific set of “relevant” pages (teleport set)

- To make this work all we need is to update the teleportation part of the PageRank formulation:

$$A_{ij} = \begin{cases} \beta M_{ij} + (1 - \beta)/|S| & \text{if } i \in S \\ \beta M_{ij} + 0 & \text{otherwise} \end{cases}$$

- $A$ is a stochastic matrix!
Social Network Algorithms

Ansh Shukla
Graph Algorithms

- **Problem**: Finding “communities” in large graphs
  - A community is any structure we’re interested by in the graph.
  - Examples of properties we might care about: overlap, triangles, density.
Personalized PageRank with Sweep

- Problem: Finding densely linked, non-overlapping communities.
- Intuition: Give a score to all nodes, rank nodes by score, and then partition the ranked list into clusters.
- What to know:

(Algorithm) Approximate Personalized PageRank –

- Frame PPR in terms of lazy random walk
- While error measure is too high
  - Run one step of lazy random walk

\[
q_u = p_u - r_u \quad \max_{u \in V} \frac{q_u}{d_u} \geq \epsilon
\]

\[
r_u^{(t+1)} = \frac{1}{2} r_u^{(t)} + \frac{1}{2} \sum_{i \rightarrow u} \frac{1}{d_i} r_i^{(t)}
\]
Personalized PageRank with Sweep

- **Problem**: Finding densely linked, non-overlapping communities.

- **Intuition**: Give a score to all nodes, rank nodes by score, and then partition the ranked list into clusters.

- **What to know:**

  - **Cut**: Set of edges with only one node in the cluster:
    \[
    \text{cut}(A) = \sum_{i \in A, j \notin A} w_{ij}
    \]

  - **Note**: This works for weighted and unweighted (set all \( w_{ij} = 1 \)) graphs.

  - Example:
    \[
    \text{cut}(A) = 2
    \]
Personalized PageRank with Sweep

• **Problem:** Finding densely linked, non-overlapping communities.

• **Intuition:** Give a score to all nodes, rank nodes by score, and then partition the ranked list into clusters.

• **What to know:**
  
  **Criterion:** *Conductance:*
  Connectivity of the group to the rest of the network relative to the density of the group

  \[
  \phi(A) = \frac{|\{(i, j) \in E; i \in A, j \not\in A\}|}{\min\{\text{vol}(A), 2m - \text{vol}(A)\}}
  \]
Personalized PageRank with Sweep

- **Problem:** Finding densely linked, non-overlapping communities.
- **Intuition:** Give a score to all nodes, rank nodes by score, and then partition the ranked list into clusters.
- **What to know:**

  **Sweep:**
  - Sort nodes in decreasing PPR score $r_1 > r_2 > \cdots > r_n$
  - For each $i$ compute $\phi(A_i = \{r_1, \ldots, r_i\})$
  - **Local minima** of $\phi(A_i)$ correspond to good clusters
Motif-based spectral clustering

• **Problem:** Finding densely linked, non-overlapping communities (as before), but changing our definition of “densely linked”.

• **Intuition:** Modify graph so edge weights correspond to our notion of density, modify conductance criteria, run PPR w/ Sweep.

• **What to know:**
  - $W_{ij}^{(M)} = \#$ times $(i, j)$ participates in the motif

![Graph G](image1)

![Weighted graph $W^{(M)}$](image2)
Motif-based spectral clustering

• **Problem:** Finding densely linked, non-overlapping communities (as before), but changing our definition of “densely linked”.

• **Intuition:** Modify graph so edge weights correspond to our notion of density, modify conductance criteria, run PPR w/ Sweep.

• **What to know:**

\[
\phi_M(S) = \frac{\#(\text{motifs cut})}{\text{vol}_M(S)}
\]
Searching for small communities (trawling)

• Problem: Finding complete bipartite subgraphs $K_{s,t}$
• Intuition: Reframe the problem as one of finding frequent itemsets: think of each vertex as a basket defined by its neighbors. Run A-priori with frequency threshold $s$ to get item sets of size $t$.
• What to know:

$$a = \{b, c, d\}$$
Graph Embeddings

- **Problem**: Want to represent nodes in graph in vector space while capturing relevant properties like graph topology.
- **Intuition**: Define a mapping from nodes to embeddings.
- Define a node similarity function (dot product)
- Optimize the parameters of the encoder so that: similarities in one representation (graph) match similarities in another (embedding)
Graph Embeddings

• **Problem:** Want to represent nodes in graph in vector space while capturing relevant properties like graph topology.

• **Intuition:** Define a mapping from nodes to embeddings.

![Diagram of graph embeddings]

- **Z** = embedding matrix
- **embedding vector for a specific node**
- Dimension/size of embeddings
- one column per node
Graph Embeddings

- **Problem:** Want to represent nodes in graph in vector space while capturing relevant properties like graph topology.

- **Intuition:**
  - Select a random walk

- $p, q$ model transition probabilities
  - $p$ ... return parameter
  - $q$ ... “walk away” parameter
Graph Embeddings

- **Problem:** Want to represent nodes in graph in vector space while capturing relevant properties like graph topology.
- **Intuition:**
  - Optimize embedding

\[
\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} \text{exp}(z_u^T z_v) - \log \left( \frac{\text{exp}(z_u^T z_v)}{\sum_{n \in V} \text{exp}(z_u^T z_n)} \right)
\]

- sum over all nodes \( u \)
- sum over nodes \( v \) seen on random walks starting from \( u \)
- predicted probability of \( u \) and \( v \) co-occurring on random walk
Graph Embeddings

- **Problem:** Want to represent nodes in graph in vector space while capturing relevant properties like graph topology.

- **Intuition:**
  - Optimize embedding

\[
\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} - \log(P(v | z_u))
\]
Large-Scale Machine Learning

Jerry Zhilin Jiang
Large-scale machine learning

Overview

• **Supervised learning**
  • given training set with labels \((x_i, y_i)\)
  • Learn the function \(f\) that predicts \(y\) given \(x\), \(f(x) = y\)
  • Why Hard? Need to generalize well to unseen data

• **Classification vs. Regression**
  • Classification: Label \(y\) belongs to a discrete set
  • Regression: Label \(y\) is continuous

• **Methods covered in this course**
  • Decision Tree
  • Support Vector Machine (SVM)
Decision Tree

- Input: \(d\) attributes (features) \(x^{(1)}, x^{(2)}, \ldots, x^{(d)}\)
  Can be numerical or categorical

- Output: \(y\) (label)
  Either numerical (regression) or categorical (classification)

- Given data point \(x_i\)
  Start from root, “drop” it down the tree until it hits a leaf node

- Make prediction accordingly after reaching the leaf node

Three problems:
- How to split?
- When to stop?
- How to predict?
How to split?

Measure the quality of potential splits based on some criterion

**Regression: Purity**  Split on node \((x^{(i)}, \nu)\), create \(D, D_L, D_R\) (parent / left child/ right child dataset)

\[
|D| \cdot Var(D) - (|D_L| \cdot Var(D_L) + |D_R| \cdot Var(D_R))
\]

**Classification: Information Gain**  \(IG(Y|X)\)
How much information about \(Y\) is contained in \(X\).

\[
IG(Y|X) = H(Y) - H(Y|X)
\]

Entropy \(H(x) = - \sum_{j=1}^{n} p_j \log p_j\)

Conditional entropy
\[
H(Y|X) = \sum_{j=1}^{n} P(X = v_j) H(Y|X = v_j)
\]
When to stop?

- When the leaf is “pure” (variance below threshold)
- When # of examples in a leaf node is too small

How to predict?

Regression
- predict average $y_i$ of examples in the leaf
- Build linear regression model on the example points

Classification
- Predict most common $y_i$ in the leaf
Building decision trees with MapReduce: **PLANET**

- Tree small (in memory), data too large to keep in memory
- Hundreds of numerical (discrete or continuous) attributes
- Target variable is numerical (i.e. regression)

Build the decision tree one level at a time

**Master Node**
Keep track of the model and decides how to grow the tree

**MapReduce**
Do the actual work on data
3 Types of MapReduce jobs:

- **Initialization** (run once first)
  - Find candidate splits ((node n, attribute $X^{(j)}$, value v))
  - Ideally divide data into similar-sized buckets
- **FindBestSplit** (run multiple times)
  - For a split node j find $X^{(j)}$ and v that **maximizes purity**
- **InMemoryBuild** (run once last)
  - If there is little data entering a tree node, Master runs an InMemoryBuild MapReduce job to grow the entire subtree below that node, including leaves
Learning Ensembles

Bagging

• Learn **multiple trees**, each using an independently sampled subset of the training data (sampled with replacement)
• Predictions from all trees are aggregated (e.g. majority vote, average) to compute the final model prediction

Improvement: Random Forests

• At each candidate split, consider only a random subset of all available features
• Avoids cases where all trees select the same few strong features (Breaks correlation between different decision trees)
• Achieves state-of-the-art results in many classification problems
SVM

Given training data
\((x_1, y_1) \ldots (x_n, y_n)\)
x: d-dimensional, real valued
\[ x_i = (x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(d)}) \]
\[ y_i = -1 \text{ or } +1 \]

\(A, B, C\): support vectors, uniquely define the decision boundary

Margin \(\gamma\): distance of closest example from the decision line (hyperplane)

Goal: maximize margin \(\gamma\), find separating hyperplane with the largest distance possible from both positive / negative point
From maximize $\gamma$ to minimize $\frac{1}{2}||w||^2$

A lying on support plane

Goal: **Maximize distance** $||AH||$

$$MA \cdot w = ||w|| \times ||AM|| \times \cos \theta$$

$$w \cdot A + b = \gamma$$

$$w \cdot M + b = 0$$

$$||AH|| = ||AM|| \times \cos \theta$$

$$= \frac{|AM \cdot w|}{||w||}$$

$$= \frac{|(A - M) \cdot w|}{||w||}$$

$$= \frac{|A \cdot w - M \cdot w|}{||w||}$$

$$= \frac{|(\gamma - b) - (-b)|}{||w||}$$

$$= \frac{\gamma}{||w||}$$

**Distance from A to H:** $\gamma$ measured in $||w||$

**Scale $\gamma$ and $||w||$ both by 2, nothing changes, thus we can either**

- Normalize $w$, i.e $||w|| = 1$, maximize $\gamma \Leftrightarrow$
- Fix margin $\gamma = 1$, minimize length of $w$

**We use the second way**
Optimization problem formalized

fix margin $\gamma = 1$, minimize length of $w$

$$
\min_w \frac{1}{2} \| w \|^2 \\
\text{s.t.} \forall i, y_i(w \cdot x_i + b) \geq 1
$$

In real world, data is often not linear separable - Introduce penalty

$$
\arg\min_{w,b} \frac{1}{2} w \cdot w + C \cdot \sum_{i=1}^{n} \max\{0,1 - y_i(w \cdot x_i + b)\}
$$

Margin

Regularization hyperparameter

Empirical loss $L$ (how well we fit training data)

Penalize mis-predicted points AND correctly predicted points that fall within the margin
Cost Function

\[ J(w, b) = \frac{1}{2} \sum_{j=1}^{d} (w^{(j)})^2 + C \sum_{i=1}^{n} \max\left\{ 0, 1 - y_i \left( \sum_{j=1}^{d} w^{(j)} x_i^{(j)} + b \right) \right\} \]

Minimizing cost function J
- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent

\[ \theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}). \]
Large-scale machine learning

Summary

- **Decision Tree**
  - Classification or Regression
  - Numerical or categorical features, usually dense
  - Complicated decision boundaries

- **Support Vector Machine (SVM)**
  - Classification (usually $y = \pm 1$)
  - High-dimensional, sparse feature space
  - Simple, linear decision boundary
Streaming Algorithms

Wensi Yin
Bloom filters - Problem

• You have a stream of ads. How to make sure a user doesn’t see the same ad multiple times?
• Naïve approach: store the ads in a hash table.
  • This takes $O(#\text{ ads})$ space!
• What if we want to use at most 100 slots of memory?
  • We can not have a deterministic answer, but we can answer it with high prob!
Bloom filters - Construction

• What if we want to use at most 100 slots of memory?
• Create a bit array B of size 100, initialized to all 0’s
• Create a hash function that hashes ads to 100 different possible buckets
• When an ad is seen, hash the ad to a bucket (say, bucket 79), and set B[79] = 1
Bloom filters - Test

• How to check whether a new incoming ad has been seen?
• Suppose the ad hashes to bucket 89
• If B[89] = 0, you know the ad has NOT been seen
• If B[89] = 1, the ad might have been seen, but we also might have seen a different ad that happened to hash to the same bucket.

• **Prob false positive:** \( 1 - \left( 1 - \frac{1}{100} \right)^m \), (m: # distinct ads seen so far)

• K number of hash functions: check if all of the k bits corresponding to the hash functions are set to 1.

• **Reasonable number** of hash functions will help **reduce** false positive prob.
Flajolet-Martin Algorithm

• Problem: a data stream consists of elements chosen from a set of size n. Maintain a count of the number of distinct elements seen so far.
• Pick a hash function \( h \) that element in set to \( \log_2 n \) bits.
• For each stream element \( a \), let \( r(a) \) be the number of trailing 0’s in \( h(a) \).
  • Record \( R = \) the maximum \( r(a) \) seen for any \( a \) in the stream.
  • Also known as the “tail length”
• Estimate of distinct elements = \( 2^R \).
• Intuitively, seeing \( r \) trailing 0s is “unusual” (prob \( 1/(2^r) \))
  • More distinct elements leads to a higher chance of seeing this “unusual” event
• If we notice this “unusual” event, our estimate should be correspondingly higher
AMS method

• Problem: Suppose a stream has elements chosen from a set of \( n \) values. Let \( m_i \) be the number of times value \( i \) occurs. **Estimate the \( k \)-th moment** which is the sum of \( m_i^k \) over all \( i \).
  
  - \( 0^{\text{th}} \) moment = number of distinct elements in the stream.
  - \( 1^{\text{st}} \) moment = sum of counts of the numbers of elements = length of the stream.
  - \( 2^{\text{nd}} \) moment = measure of how uneven the distribution is.

• Algorithm for \( 2^{\text{nd}} \) moment:
  
  - Assume stream seen so far has \( n \) elements
  - Pick a random starting and let the chosen time have element \( a \) in the stream.
  - Let \( X = \) # times \( a \) is seen in the stream from that point onward
  - Estimate of \( 2^{\text{nd}} \) moment = \( n(2X - 1) \)

• Application:
  
  - \( 2^{\text{nd}} \) moment can be used to estimate self-join size in database.
Computational Advertising

Stefanie
Advertising: Bipartite Matching

$M = \{(1,a),(2,b),(3,d)\}$ is a matching

Cardinality of matching $= |M| = 3$
Advertising: Online Algorithms and Competitive Ratio

- Question: How to find a maximum matching for a given bipartite graph
- Polynomial offline algorithm exists, but what's the best we can do in online setting?

**Competitive ratio** = 

\[ \min_{\text{all possible inputs } I} \left( \frac{|M_{\text{greedy}}|}{|M_{\text{opt}}|} \right) \]

(greedy's worst performance over all possible inputs I)

- In maximization problem, competitive ratio <= 1
- In minimization problem, competitive ratio >= 1
- **Greedy bipartite matching algorithm**: competitive ratio = 1/2.
  - Easy to find examples, proofs are more difficult
Advertising: Adwords and Click Through Rate

- Adwords problem is example of online algorithm
  
  **Instead of sorting advertisers by bid, sort by expected revenue**

<table>
<thead>
<tr>
<th>Advertiser</th>
<th>Bid</th>
<th>CTR</th>
<th>Bid * CTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>$0.75</td>
<td>2%</td>
<td>1.5 cents</td>
</tr>
<tr>
<td>C</td>
<td>$0.50</td>
<td>2.5%</td>
<td>1.125 cents</td>
</tr>
<tr>
<td>A</td>
<td>$1.00</td>
<td>1%</td>
<td>1 cent</td>
</tr>
</tbody>
</table>

**Challenges:**

- CTR of an ad is unknown
- Advertisers have limited budgets and bid on multiple queries
Advertising: Greedy vs BALANCE Algorithm

• Simplified setting:
  • There is 1 ad shown for each query
  • All advertisers have the same budget $B$
  • All ads are equally likely to be clicked
  • Value of each ad is the same ($=1$)
• Greedy Algorithm: Pick any advertiser who has a bid for query
  • Competitive ratio is 1/2.
• **BALANCE Algorithm**: Pick advertiser with largest unspent budget
  • Competitive ratio is $(1-1/e) = 0.63$
  • No online algorithm can do better!
Advertising: 2 Case Analysis

Queries allocated to $A_1$ in optimal solution
Queries allocated to $A_2$ in optimal solution
Opt revenue = 2B
Balance revenue = 2B-x

We claim $x \leq B/2$
=> Competitive Ratio = 3/4.
Advertising: Generalized BALANCE Algorithm

• Generalized Scenario:
  • Arbitrary bids and arbitrary budgets

• BALANCE algorithm has arbitrary bad competitive ratio
  • competitive ratio -> 0

• Generalized BALANCE: consider query \( q \), bidder \( i \)
  • Bid = \( x_i \)
  • Budget = \( b_i \)
  • Amount spent so far = \( m_i \)
  • Fraction of budget left over \( f_i = \frac{1}{b_i} - m_i \)
  • Define \( \psi_i(q) = x_i(1-e^{-f_i}) \)

• Allocate query \( q \) to bidder \( i \) with largest value of \( \psi_i(q) \)

• Same competitive ratio \( (1-1/e) = 0.63 \)
Learning Through Experimentation

Baige(Alice) Liu
Learning Through Experimentation

• Take action, get reward, learn from that reward.
• Approach: formalize as a Multiarmed Bandits. Take action = pull an arm.
Multiarmed Bandits

• K-armed bandit: $|\text{action}| = K$.

• Each arm $a$ wins (reward = 1) with fixed (unknown) probability $\mu_a$, and loses (reward = 0) with fixed (unknown) probability $1 - \mu_a$.

• Want to maximize total reward, but need information about (unknown) $\mu_a$.

• Every time we pull $a$, we learn a bit about $a$, so we can estimate $\mu_a$ (denoted as $\hat{\mu}_a$).
Bandit Algorithm: Greedy and Epsilon-Greedy

• Tradeoff between exploration and exploitation.
• Exploration: pull arm haven’t tried before. Exploitation: pull arm with current highest $\mu_a$.
• Greedy algorithm takes action with highest average reward based on samples seen so far ($\mu_a$). But it does not explore sufficiently.
• Epsilon-Greedy takes a random $a$ with a decaying probability $\varepsilon_t (O(\frac{1}{t}))$, and it takes the same action that Greedy would take with probability $1 - \varepsilon_t$. During exploration time, it selects random $a$ equally likely, which is suboptimal.
Bandit Algorithm: $UCB_1$

• Balances exploration and exploitation by taking confidence into consideration.

• A confidence interval is a range of values within which we are sure the mean lies with a certain probability.

• Let $m_a = \text{number of times } a \text{ is pulled, } \delta = \text{given confidence level}.$

• Then, confidence interval $b = \max(\mu_a | \delta) - \min(\mu_a | \delta) = 2 \sqrt{\frac{2 \ln T}{m_a}}.$

• $UCB(a) = \hat{\mu}_a + \alpha \sqrt{\frac{2 \ln T}{m_a}}.$
Bandit Algorithm: $UCB_1$

• The accuracy of $\hat{\mu}_a$ is dependent on how many times we have tried $a$: trying $a$ too few times means our estimate of $\mu_a$ could be very off from the true value $\mu_a$, which means it has a large confidence interval. This interval shrinks as we try $a$ more often.

• Strategy: try arm $a$ with the highest upper bound on its confidence interval, i.e., action as good as possible given the available evidence. It is called an optimistic policy.

• $UCB(a) = \hat{\mu}_a + \alpha \sqrt{\frac{2 \ln T}{m_a}}$. 