MapReduce and Frequent Itemsets Mining

Kushaagra Goyal
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
  - k4:v
  - k1:v, k3:v

- **Group by Key:**
  - Grouped:
    - k1:v, v, v, v
    - k2:v
    - k3:v, v
    - k4:v, v, 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

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

Pass 1
- Hash table for pairs
- Item counts

Pass 2
- Bitmap
- Frequent items
- 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

Heather Blundell
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 ⇒
  Become candidate pair:
  - \( 1 - (1 - s^r)^b \)
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 \(\Rightarrow (d_1, d_2, p_1^r, p_2^r)\)-sensitive family
- Lowers all probabilities

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

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

We can try to make \(p_1 \rightarrow 1\) (lower false negatives) and \(p_2 \rightarrow 0\) (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 $\text{max}$ distance (worst case), $\text{min}$ distance (best case), or $\text{average}$ 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

Qijia Jiang
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

- $M = UΣV^T$
  - $U^TU = I, V^TV = I, Σ$ diagonal
  - Best low-rank approximation (singular value thresholding)

- Algorithm
  - Find $Σ, V$
    - Find eigenpairs of $M^TM \rightarrow (D, V)$
    - $Σ$ 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 evec
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
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:** \( r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}} \)

- Remove baseline estimate and only model rating deviations from baseline estimate, so that we’re not affected by user/item bias
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

Jessica Su
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

- \( r_a = 0.8(r_y/2 + r_m) + 0.2/3 \)
- \( r_y = 0.8(r_y/2 + r_a/2) + 0.2/3 \)
- \( r_m = 0.8 \left( r_a / 2 \right) + 0.2/3 \)
PageRank

- **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: \[ r = A \cdot r \]
Hubs and Authorities

- Similar to PageRank
- Every webpage gets two scores: an “authority” score, which measures the quality of the webpage, and a “hub” score, which measures how good it is at linking to good webpages
- Mutually recursive definition:
  - Good hubs link to good authorities
  - Good authorities are linked to by good hubs
Hubs and Authorities

- Adjacency matrix \( A \ (N \times N) \): \( A_{ij} = 1 \) if \( i \rightarrow j \), 0 otherwise

- Set: \( a_i = h_i = \frac{1}{\sqrt{n}} \)

Repeat until convergence:

- \( h = A \cdot a \) (sum contributions of outbound links)
- \( a = A^T \cdot h \) (sum contributions of inbound links)

Normalize \( a \) and \( h \)
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

\[
\begin{align*}
 r_u^{(t+1)} &= \frac{1}{2}r_u^{(t)} + \frac{1}{2} \sum_{i \rightarrow u} \frac{1}{d_i} r_i^{(t)} \\
 q_u &= p_u - r_u \\
 \max_{u \in V} \frac{q_u}{d_u} \geq \epsilon
\end{align*}
\]
• **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.

  ![Diagram](image)
• **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.

\[
\phi(A) = \frac{\left| \{(i, j) \in E; i \in A, j \notin A\} \right|}{\min(\text{vol}(A), \ 2m - \text{vol}(A))}
\]
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**:
  - $W^{(M)}_{ij} = \# \text{times } (i, j) \text{ participates in the motif}$

[Diagram showing graphs $G$ and $W^{(M)}$]
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:**

\[ a = \{b, c, d\} \]
Detecting overlapping communities with AGM

• **Problem:** Understand structural properties of individual communities given a graph where communities overlap (i.e. vertex may belong to multiple communities).

• **Intuition:** Beforehand, decide (random, parameterized) model for how communities are created. Pick parameters for the model which maximize the chance that it created the community you’re looking at. Infer structural properties from maximized parameters.
Detecting overlapping communities with AGM

• **Problem**: Understand structural properties of individual communities given a graph where communities overlap.

• **Intuition**: Decide model for how communities are created. Maximize likelihood of community analyzed.

• **What to know**: AGM model parameterized by:
  
  • \( B(V, C, M, \{p_c\}) \)
Detecting overlapping communities with AGM

• **Problem:** Understand structural properties of individual communities given a graph where communities overlap.

• **Intuition:** Decide model for how communities are created. Maximize likelihood of community analyzed.

• **What to know:** Setting up a likelihood expression given a model.

$$\arg \max_{B(V,C,M,\{p_c\})} \sum_{u,v \in E} \log P(u,v) + \sum_{u,v \notin E} \log(1 - P(u,v))$$
Detecting overlapping communities with AGM

- **Problem:** Understand structural properties of individual communities given a graph where communities overlap.
- **Intuition:** Decide model for how communities are created. Maximize likelihood of community analyzed.
- **What to know:** Computational simplification of individual community affiliations boosts instead of global ones.

\[ P_A(u, v) = 1 - \exp(-F_{uA} \cdot F_{vA}) \]
Detecting overlapping communities with AGM

- **Problem:** Understand structural properties of individual communities given a graph where communities overlap.
- **Intuition:** Decide model for how communities are created. Maximize likelihood of community analyzed.
- **What to know:** Setting up gradient descent for the log-likelihood maximization problem.
Large-Scale Machine Learning

Wanzi Zhou
Large-scale machine learning

High Level

• **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

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

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

Given training data

\[(x_1, y_1) \ldots (x_n, y_n)\]

\(x: d\)-dimensional, real valued

\[x_i = (x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(d)})\]

\[y_i = -1 \text{ or } +1\]

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

**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 $w$

$$\min_w \frac{1}{2} \| w \|^2$$

$$s.t. \forall i, y_i (w \cdot x_i + b) \geq 1$$

Data not 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 parameter
Empirical loss $L$ (how well we fit training data)

Penalize mispredicted points as well as correctly predicted points
But within the margin
How solve in large scale data

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

Minimize cost function J

- Batch Gradient descent
- Stochastic gradient descent
- Minibatch gradient descent
Decision Tree

- Input: \(d\) attributes/features
  \(x^{(1)}, x^{(2)}, \ldots, x^{(d)}\)
  Numerical / categorical

- Output variable: \(Y\)
  Either numerical (regression) or categorical (classification)

- “Drop” \(x_i\) down the tree until it hits a leaf node

Three problems:
1. How to split
2. When to stop
3. How to predict
How to split

measure the quality of a split based on some criterion

**Regression: Purity** Split on node \((x^{(i)}, v)\), 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** \(IG(Y \mid X)\)

How much information about \(Y\) is contained in \(X\).

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

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

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

1. Leaf is pure (variance is small)
2. # examples in a leaf is small

How to predict (many options)

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 map reduce: **PLANET**

- Tree small (in memory), data too large to keep in memory
- Hundreds of numerical attributes
- Target variable numerical

Build the decision tree one level at a time

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

**MapReduce**
Given a set of split candidates compute their quality
3 Types of MapReduce jobs:

- **Initialization** (run once)
  - Find candidate splits *(node n, attribute \( X^{(j)} \), value v)*
  - Approximate equivalent depth in data
- **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
Bagging

idea:
• Learn **multiple trees** over independent samples of the training data
• Predictions from each tree are averaged to compute the final model prediction

Sample training data with replacement

Feature bagging: random sample features for different trees (break correlation between different decision trees)
Streaming Algorithms

Jessica Su
Reservoir sampling

• How to pick a random element of an array?
Reservoir sampling

• How to pick a random element of an array?
  • Choose a random integer, and pick the element at that index
Reservoir sampling

• How to pick a random element of an array?
  • Choose a random integer, and pick the element at that index

• How to pick $s$ random elements from an infinite stream of elements?
Reservoir sampling

• How to pick a random element of an array?
  • Choose a random integer, and pick the element at that index

• How to pick $s$ random elements from an infinite stream of elements?
  • What does this mean? At any given point in the stream, you should be able to output an “answer.”
Reservoir sampling

• How to pick a random element of an array?
  • Choose a random integer, and pick the element at that index

• How to pick $s$ random elements from an infinite stream of elements?
  • What does this mean? At any given point in the stream, you should be able to output an “answer.”
  • Maintain a set of $s$ elements that may be updated every time a new element is added to the stream
  • At any given point, each element that has been seen so far should be equally likely to be in the set of $s$ elements.
Reservoir sampling

• Save the first $s$ elements
Reservoir sampling

- Save the first $s$ elements
- Suppose we have seen the first $n - 1$ elements, and now the $n$th element arrives
Reservoir sampling

• Save the first $s$ elements
• Suppose we have seen the first $n - 1$ elements, and now the $n$th element arrives
• With probability $s/n$, keep the $n$th element, and replace a randomly chosen saved element with the $n$th element.
Reservoir sampling

• Save the first $s$ elements
• Suppose we have seen the first $n - 1$ elements, and now the $n$th element arrives
• With probability $s/n$, keep the $n$th element, and replace a randomly chosen saved element with the $n$th element.
• Otherwise, discard the $n$th element.
Reservoir sampling

• Save the first $s$ elements
• Suppose we have seen the first $n - 1$ elements, and now the $n$th element arrives
• With probability $s/n$, keep the $n$th element, and replace a randomly chosen saved element with the $n$th element.
• Otherwise, discard the $n$th element.
• **Induction:** After $n$ elements, the sample contains each element seen so far with probability $s/n$
Bloom filters

• You have a stream of ads. How to make sure a user doesn’t see the same ad multiple times?
Bloom filters

• 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.
Bloom filters

- 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!
Bloom filters

• 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?
Bloom filters

• 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
Bloom filters

• 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
Bloom filters

• 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

• How to check whether an ad has been seen?
Bloom filters

• How to check whether an ad has been seen?
• Suppose the ad hashes to bucket 79
Bloom filters

• How to check whether an ad has been seen?
• Suppose the ad hashes to bucket 79
• If B[79] = 0, you know the ad has NOT been seen
Bloom filters

• How to check whether an ad has been seen?
• Suppose the ad hashes to bucket 79
• If B[79] = 0, you know the ad has NOT been seen
• If B[79] = 1, the ad might have been seen, but we also might have seen a different ad that happened to hash to the same bucket.
Bloom filters

• How to check whether an ad has been seen?
• Suppose the ad hashes to bucket 79
• If $B[79] = 0$, you know the ad has NOT been seen
• If $B[79] = 1$, the ad might have been seen, but we also might have seen a different ad that happened to hash to the same bucket.
• Reduce false positives by using multiple hash functions and checking if ALL of the bits corresponding to the hash functions are 1.
Computational Advertising

Hiroto Udagawa
Advertising: Bipartite Matching

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

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?

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

(greedy’s worst performance over all possible inputs !)

• 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

Balance allocation

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 \( \rightarrow 0 \)

• Generalized BALANCE: consider query \( q \), bidder \( i \)
  • Bid = \( x_i \)
  • Budget = \( b_i \)
  • Amount spent so far = \( m_i \)
  • Fraction of budget left over \( f_i = 1-m_i/b_i \)
  • 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

Chang Yue
Learning Through Experimentation

• Take action, get reward, learn from that reward.
• Doing this repeatedly is called Learning Through Experimentation.
• 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 \( \hat{\mu}_a \).
• Greedy algorithm takes action with highest average reward based on samples seen so far (\( \hat{\mu}_a \)). But it does not explore sufficiently.
• Epsilon-Greedy takes a random \( a \) with a decaying probability \( \varepsilon_t \left( O \left( \frac{1}{t} \right) \right) \), 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 $m_a = \text{number of times } a \text{ is pulled}, \delta = \text{given confidence level}.$
- Hoeffding’s Inequality: $P(\mu_a \geq \hat{\mu}_a + b') \leq e^{-2b'\mu^2} m_a = \delta.$
- Usually set $\delta = T^{-4},$ where $T$ is the total number of actions have taken.
- $b' = \max(\mu_a|\delta) - \hat{\mu}_a = \hat{\mu}_a - \min(\mu_a|\delta).$
- Then, confidence interval $b = \max(\mu_a|\delta) - \min(\mu_a|\delta) = 2b' = 2\sqrt{\frac{2\ln T}{m_a}}.$
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.

- $UCB_1$: 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.

- $UCB(a) = \hat{\mu}_a + \alpha b' = \hat{\mu}_a + \alpha \sqrt{\frac{2 \ln T}{m_a}}$. 
Optimizing Submodular Functions

Praty Sharma
Set Cover Problem

- **Suppose we are given a set of documents** $D$
  - Each document $d$ covers a set $X_d$ of words/topics/named entities $W$

- **For a set of documents** $A \subseteq D$ we define
  $$F(A) = \left| \bigcup_{d \in A} X_d \right|$$

- **Goal:** We want to
  $$\max_{|A| \leq k} F(A)$$

- **Note:** $F(A)$ is a set function: $F(A) : \text{Sets} \rightarrow \mathbb{N}$
Simple Greedy Heuristic

**Simple Heuristic: Greedy Algorithm:**

• Start with $A_0 = \{ \}$
• For $i = 1 \; \ldots \; k$
  • Find set $d$ that $\max F(A_{i-1} \cup \{d\})$
  • Let $A_i = A_{i-1} \cup \{d\}$

**Example:**

• Eval. $F(\{d_1\})$, $\ldots$, $F(\{d_m\})$, pick best (say $d_1$)
• Eval. $F(\{d_1\} \cup \{d_2\})$, $\ldots$, $F(\{d_1\} \cup \{d_m\})$, pick best (say $d_2$)
• Eval. $F(\{d_1, d_2\} \cup \{d_3\})$, $\ldots$, $F(\{d_1, d_2\} \cup \{d_m\})$, pick best
• And so on...

\[
F(A) = \left| \bigcup_{d \in A} X_d \right|
\]
Approximation Guarantee

- **Greedy** produces a solution $A$ where: $F(A) \geq (1 - \frac{1}{e}) \times OPT$  
  ($F(A) \geq 0.63 \times OPT$)
  
  [Nemhauser, Fisher, Wolsey ‘78]

- Claim holds for functions $F(\cdot)$ with 2 properties:
  - **$F$ is monotone**: (adding more docs doesn’t decrease coverage)
    if $A \subseteq B$ then $F(A) \leq F(B)$ and $F(\{\})=0$
  - **$F$ is submodular**: 
    adding an element to a set gives less improvement than adding it to one of its subsets
Submodularity Definition

- **Definition**

  \[ F(\cdot) \text{ is submodular: } A \subseteq B \]

  \[
  F(A \cup d) - F(A) \geq F(B \cup d) - F(B)
  \]

  Gain of adding \( d \) to a small set

  Gain of adding \( d \) to a large set

- **Natural example:**
  - Sets \( d_1, \ldots, d_m \)
  - \( F(A) = |\bigcup_{i \in A} d_i| \)
    (size of the covered area)
  - **Claim:**
    \( F(A) \) is submodular!
Submodularity & Concavity

• **Marginal gain:**
  \[
  \Delta_F(d|A) = F(A \cup d) - F(A)
  \]

• **Submodular:**
  \[
  F(A \cup d) - F(A) \geq F(B \cup d) - F(B) \\
  A \subseteq B
  \]

• **Concavity:**
  \[
  f(a + d) - f(a) \geq f(b + d) - f(b) \\
  a \leq b
  \]
Submodularity: Useful Fact

• Let $F_1 \ldots F_m$ be submodular and $\lambda_1 \ldots \lambda_m > 0$ then $F(A) = \sum_i^m \lambda_i F_i(A)$ is submodular
  • Submodularity is closed under non-negative linear combinations!

• This is an extremely useful fact:
  • Average of submodular functions is submodular: $F(A) = \sum_i P(i) \cdot F_i(A)$
  • Multicriterion optimization: $F(A) = \sum_i \lambda_i F_i(A)$
Probabilistic Set Cover

• **Document coverage function:**
  \[ \text{cover}_d(c) = \text{probability document } d \text{ covers concept } c \]
  - \( \text{Cover}_d(c) \) can also model how relevant is concept \( c \) for user \( u \)

• **Set coverage function:**
  \[ \text{cover}_A(c) = 1 - \prod_{d \in A} (1 - \text{cover}_d(c)) \]
  - Prob. that at least one document in \( A \) covers \( c \)

• **Objective:**
  \[ \max_{A:|A| \leq k} F(A) = \sum_{c} w_c \text{ cover}_A(c) \]
Optimizing $F(A)$

$$\max_{A : |A| \leq k} F(A) = \sum_c w_c \text{ cover}_A(c)$$

- The objective function is also **submodular**
  - Intuitive **diminishing returns** property
  - Greedy algorithm leads to a $(1 - 1/e) \approx 63\%$ approximation, i.e., a **near-optimal** solution
Speeding up Greedy

**Greedy**

- **Greedy algorithm is slow!**
  - At each iteration we need to re-evaluate marginal gains of all remaining documents
  - Runtime $O(|D| \cdot K)$ for selecting $K$ documents out of the set of $D$ of them

Marginal gain:

$F(A \cup x) - F(A)$

Add document with highest marginal gain
Speeding up Greedy

• **In round** \(i\): So far we have \(A_{i-1} = \{d_1, \ldots, d_{i-1}\}\)
  • Now we pick \(d_i = \arg \max_{d \in V} F(A_{i-1} \cup \{d\}) - F(A_{i-1})\)
  • Greedy algorithm **maximizes the “marginal benefit”**
    \[
    \Delta_i(d) = F(A_{i-1} \cup \{d\}) - F(A_{i-1})
    \]

• **By submodularity property:**
  \[
  F(A_i \cup \{d\}) - F(A_i) \geq F(A_j \cup \{d\}) - F(A_j) \text{ for } i < j
  \]

• **Observation:** By submodularity:
  For every \(d \in D\)
  \[
  \Delta_i(d) \geq \Delta_j(d) \text{ for } i < j \text{ since } A_i \subseteq A_j
  \]

• **Marginal benefits** \(\Delta_i(d)\) only shrink!
  (as \(i\) grows)
Lazy Greedy

**Idea:**
- Use $\Delta_i$ as upper-bound on $\Delta_j$ ($j > i$)

**Lazy Greedy:**
- Keep an ordered list of marginal benefits $\Delta_i$ from previous iteration
- Re-evaluate $\Delta_i$ only for top element
- Re-sort and prune

Upper bound on Marginal gain $\Delta_1$

```
A_1 = \{a\}
```

```
\[ F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B) \quad A \subseteq B \]
```
Lazy Greedy

• **Idea:**
  - Use $\Delta_i$ as upper-bound on $\Delta_j$ ($j > i$)

• **Lazy Greedy:**
  - Keep an ordered list of marginal benefits $\Delta_i$ from previous iteration
  - Re-evaluate $\Delta_i$ only for top element
  - Re-sort and prune

\[
F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)
\]
Lazy Greedy

**Idea:**
- Use $\Delta_i$ as upper-bound on $\Delta_j$ ($j > i$)

**Lazy Greedy:**
- Keep an ordered list of marginal benefits $\Delta_i$ from previous iteration
- Re-evaluate $\Delta_i$ only for top element
- Re-sort and prune

\[
F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B) \quad A \subset B
\]
Summary so far

- **Summary:**
  - Diversity can be formulated as a set cover
  - Set cover is a submodular optimization problem
  - Can be (approximately) solved using greedy algorithm
  - Lazy-greedy gives significant speedup

![Graph showing running time vs. number of blogs selected]

Lower is better

- **x-axis:** number of blogs selected
- **y-axis:** running time (seconds)

- **Exhaustive search (all subsets)**
- **Naive greedy**
- **Lazy**