Hybrid Filtering

Recommendations!

Fun Fact

-Netflix sponsored a competition, offering a grand prize of $1,000,000 to the team that could take an offered dataset of over 100 million movie ratings and return recommendations that were 10% more accurate than those offered by the company's existing recommender system

Filtering

- Collaborative
  - Uses user similarities based on rating patterns
  - Incorporates social network analysis through a similarity graph
  - Considers user influence through multiple centrality measures:
    - PageRank (overall influence)
    - Betweenness Centrality (bridge users)
    - Eigenvector Centrality (connection to influential users)
- Content-Based
  - Uses item features to group similar items
  - Considers user's rating history to predict preferences for new items
  - Helps address the "cold start" problem when collaborative data is sparse

Key Features

Adaptive Weighting

- Dynamically adjust the importance of collaborative vs. content-based recommendations
- Adapt based on user's position in the social network
- More content-based for bridge users (high betweenness)
- More collaborative for influential users (high PageRank)

Social Network

- Maintain a similarity graph of users
- Use Pearson correlation to measure user similarities
- Provide detailed analysis of user connections and influence

Goal: Create a sorted list of a users top 5 recommendations!

Page Rank

```
function CalculatePageRank(graph):
    n = graph.numberOfNodes
    pageRank = initializeMap(n, 1.0/n)

    for iteration = 1 to maxIterations:
    newRank = initializeMap(n, (1 - dampingFactor) / n)
    maxChange = 0

    for each node in graph:
    incomingNodes = graph.getIncomingNeighbors(node)

for each neighbor in incomingNodes:
        // Add weighted contributions from incoming edges
        weight = graph.getEdgeWeight(neighbor, node)
        outDegree = graph.getOutDegree(neighbor)
        newRank[node] +=
        dampingFactor * pageRank[neighbor] * weight / outDegree

    maxChange = max(maxChange, |newRank[node] - pageRank[node]|)

    pageRank = newRank
    if maxChange < epsilon:
    break

    return normalizeScores(pageRank)
```

Brandes Algorithm

```
function CalculateBetweennessCentrality(graph):
        betweenness = initializeMap(graph.nodes, 0)

        for each source in graph.nodes:
        // Step 1: Calculate shortest paths using BFS
        distances = initializeMap(graph.nodes, INFINITY)
        paths = initializeMap(graph.nodes, 0)
        stack = []
        queue = Queue()

        distances[source] = 0
        paths[source] = 1
        queue.push(source)

        while not queue.empty():
        vertex = queue.pop()
        stack.push(vertex)

        for each neighbor in graph.getNeighbors(vertex):
                // Path discovery
                if distances[neighbor] == INFINITY:
                distances[neighbor] = distances[vertex] + 1
                queue.push(neighbor)

                // Path counting
                if distances[neighbor] == distances[vertex] + 1:
                paths[neighbor] += paths[vertex]

        // Step 2: Accumulate dependencies
        dependencies = initializeMap(graph.nodes, 0)

        while not stack.empty():
        vertex = stack.pop()
        for each neighbor in graph.getNeighbors(vertex):
                if distances[neighbor] == distances[vertex] + 1:
                dependency = (paths[vertex] / paths[neighbor])
* (1 + dependencies[neighbor])
                dependencies[vertex] += dependency
                betweenness[vertex] += dependency

        return normalizeScores(betweenness)
```

Von Mises Iteration

```
function CalculateEigenvectorCentrality(graph, maxIterations=100,
epsilon=1e-8):
        n = graph.numberOfNodes
        eigenvector = initializeMap(n, 1.0/n)  // Initial guess

        for iteration = 1 to maxIterations:
        newVector = initializeMap(n, 0)
        maxChange = 0

        // Power iteration
        for each node in graph:
        for each neighbor in graph.getNeighbors(node):
                weight = graph.getEdgeWeight(node, neighbor)
                newVector[node] += weight * eigenvector[neighbor]

// Normalize to prevent numerical overflow
        magnitude = sqrt(sum(newVector[i]² for i in nodes))
        if magnitude == 0: break

        // Update values and check convergence
        for each node in graph:
        newVector[node] /= magnitude
        maxChange = max(maxChange, |newVector[node] -
eigenvector[node]|)

        eigenvector = newVector
        if maxChange < epsilon:
        break

        return normalizeScores(eigenvector)
```

Runtime (Hideos)

PageRank - $O(k\,(n + e))$

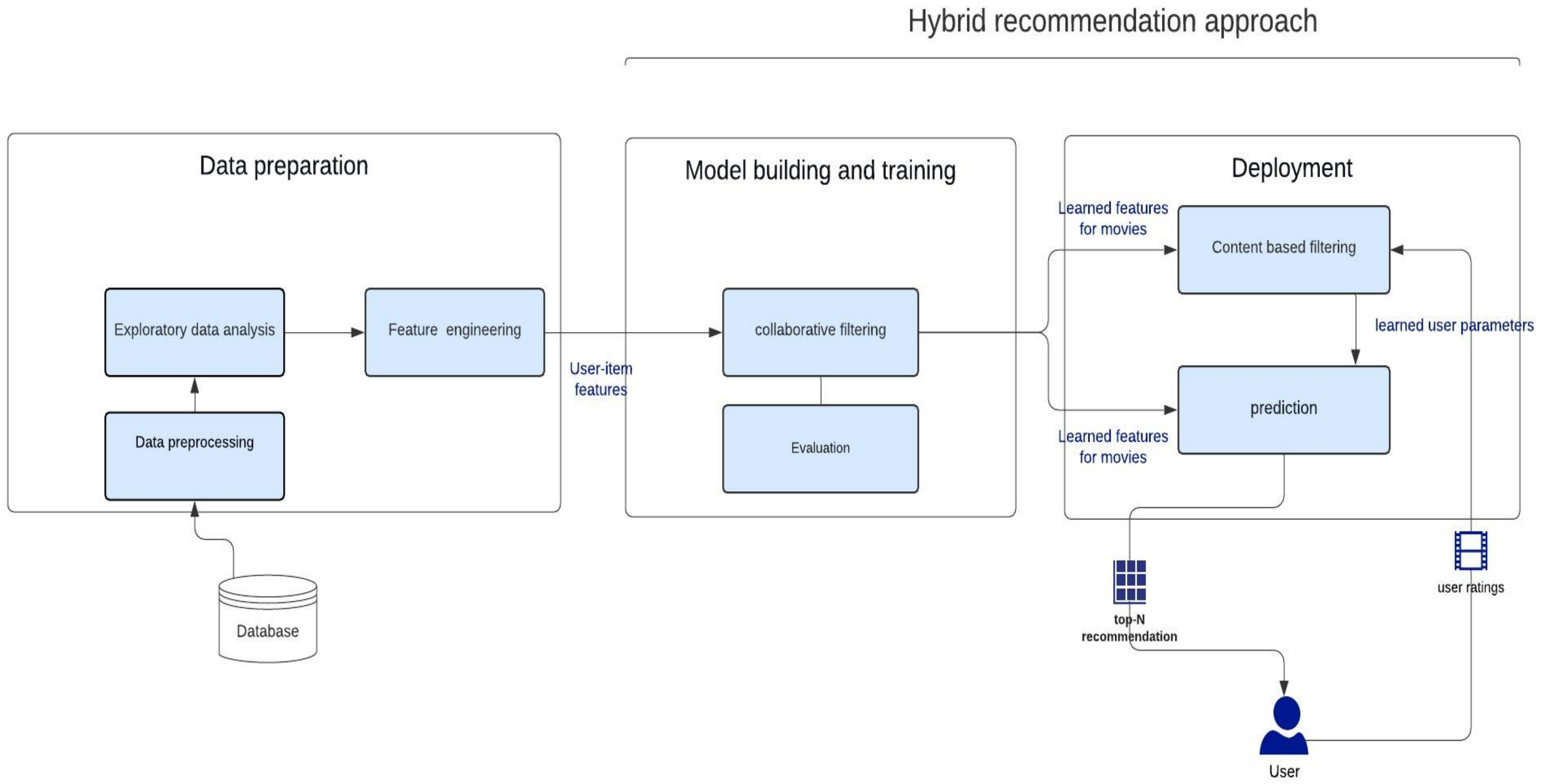
Betweenness - $O(n^2 + ne)$

Eigenvector - $O(k\,e)$

-K is based on graph convergence which I don't fully understand, but it has to do with the structure of the graph. Sparse or dense makes a huge difference depending on the algorithm.

Roadmap

SYED MUHAMMAHD HAMZA



Important Functions

```
double getDirectSimilarity(const std::string& from, const std::string& to) const;
double getPathSimilarity(const std::string& from, const std::string& to) const;
double getMultiPathSimilarity(const std::string& from, const std::string& to) const;
std::vector<std::pair<std::string, double>> getTopSimilarVertices(
```

- Open to suggestions!