Mining Massive Data Sets Endterm Report

1st Luong Canh Phong
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0036@student.tdtu.edu.com

2nd Cao Nguyen Thai Thuan
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0092@student.tdtu.edu.com

3rd Tang Minh Thien An
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0075@student.tdtu.edu.com

4th Truong Tri Phong
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0167@student.tdtu.edu.com

5th Instructor: Nguyen Thanh An
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
nguyenthanhan@tdtu.edu.com

***Abstract*—This project implements and evaluates key techniques in mining massive datasets. It covers hierarchical agglomerative clustering of string shingles using Jaccard distance; PySpark-based linear regression for gold price prediction; CUR decomposition for feature dimensionality reduction on gold price data; and PageRank analysis of the `it.tdtu.edu.vn` web graph using PySpark. The work demonstrates practical applications and provides insights into processing large-scale data.**

I. Introduction

The increasing volume of data requires efficient mining techniques. This project implements and analyzes four core algorithms: (1) hierarchical agglomerative clustering for non-Euclidean text data, using 4-shingles and Jaccard distance on alphabetical strings; (2) PySpark-based linear regression to predict Vietnamese gold prices from historical data; (3) CUR decomposition to reduce the dimensionality of gold price features (from 10 to 5) and assess its impact on regression; and (4) PageRank, implemented in PySpark, to identify influential pages within the `it.tdtu.edu.vn` web graph. Python and PySpark are utilized throughout. This report details the methodologies, implementations, and experimental results for each task.

II. First Task: Hierarchical clustering in non-Euclidean spaces

A. Overview of Agglomerative Hierarchical Clustering (AHC)

This task implements the AHC algorithm to group character strings based on similarity in shingles (4-character tokens) and Jaccard distance. The goal is to divide a large dataset (about 10,000 random strings) into a predefined number of clusters.

The main method involves calculating the clustroid (the representative sample with the smallest sum of squared distances to other samples in the cluster) and determining the distance between clusters based on the Jaccard distance between their clustroids. The algorithm gradually merges the closest pairs of clusters until the desired number of clusters is reached, using a heap for optimization.

B. Implementation of the Algorithm

The implemented AHC algorithm operates on a dataset of shingle sets (derived from alphabetical strings) to partition them into a specified number of distinct groups. This bottom-up approach initiates by treating each shingle set as an individual cluster. The core of the algorithm lies in its iterative merging strategy: in each step, it identifies the two "closest" active clusters and combines them into a single, new cluster. This process is guided by two key definitions: firstly, each cluster is represented by a clustroid, which is the member sample that minimizes the sum of squared Jaccard distances to all other members within that cluster. Secondly, the dissimilarity (or "distance") between any two clusters is determined by the Jaccard distance calculated directly between their respective clustroids. A min-priority queue (heap) is employed to efficiently manage and retrieve the closest pair of clusters at each iteration. The merging continues until the number of active clusters reduces to the desired target, yielding a final set of clusters based on the principle of grouping entities with the most similar representative points. Performance is enhanced by caching Jaccard distances between individual samples.

1) `fit(self, input_shingle_sets, num_target_clusters=None)`:

Purpose: This is the main orchestrating method for the entire clustering process.

Inputs: A list of `input_shingle_sets` (data samples) and an optional `num_target_clusters` (the desired number of final clusters).

Outputs: A list of lists, where each inner list contains the original indices of samples belonging to a final cluster.

Operation: It initializes each sample as a distinct cluster. It then iteratively identifies the pair of currently active clusters that are "closest" (based on inter-clustroid Jaccard distance), merges them into a new cluster. Then proceeds to recalculate the new cluster's representative clustroid, and updates the set of active clusters and their pairwise distances (managed via a min-priority heap). This merging process continues until the number of active clusters equals num_target_clusters.

2) `_calculate_clustroid_strictly(self, cluster_original_sample_indices)`:

Purpose: To identify the most representative sample (clustroid) within a given cluster.

Input: `cluster_original_sample_indices` (a list/tuple of original sample indices that form the cluster).

Output: The original sample index of the calculated clustroid.

Operation: This method determines the clustroid by finding the sample within the input cluster that has the minimum sum of squared Jaccard distances to all other samples in that same cluster. It exhaustively checks each member as a potential clustroid.

3) `_get_inter_cluster_distance_centroid(-self, internal_cluster_id1, internal_cluster_id2)`:

Purpose: To compute the dissimilarity (distance) between two active clusters.

Input: The `internal_cluster_ids` of two active clusters.

Output: The Jaccard distance value between the clustroids of these two clusters.

Operation: It retrieves the pre-calculated clustroids for the two input clusters (using their internal IDs) and then computes the Jaccard distance directly between these two representative clustroid samples. This value dictates which clusters are considered for merging.

4) `_get_cached_sample_distance(self, original_sample_idx1, original_sample_idx2)`:

Purpose: To efficiently retrieve or compute the Jaccard distance between two original data samples, utilizing a cache.

Input: The original indices (`original_sample_idx1`, `original_sample_idx2`) of two data samples.

Output: The Jaccard distance between the two specified samples.

Operation: It first checks if the distance for this pair of samples has already been computed and stored in `self._sample_distance_cache`. If so, it returns the cached value. Otherwise, it computes the Jaccard distance using the provided `self.distance_metric` (e.g., `jaccard_distance` function), stores it in the cache for future use, and then returns it.

5) `get_final_cluster_details(self)`:

Purpose: To retrieve detailed information about the final clusters after the fit method has completed.

Input: None (it operates on the state of the object after fit).

Output: A list of dictionaries, where each dictionary contains details for one final cluster, including its internal ID, member original indices, the shingle sets of its members, its clustroid's original sample index, and the clustroid's shingle set.

Operation: It iterates through the set of `self.active_cluster_ids` (which represent the final clusters), and for each, it compiles the relevant information from `self.cluster_members_map` and `self.clustroids_map`.

C. Experimental Results and Evaluation

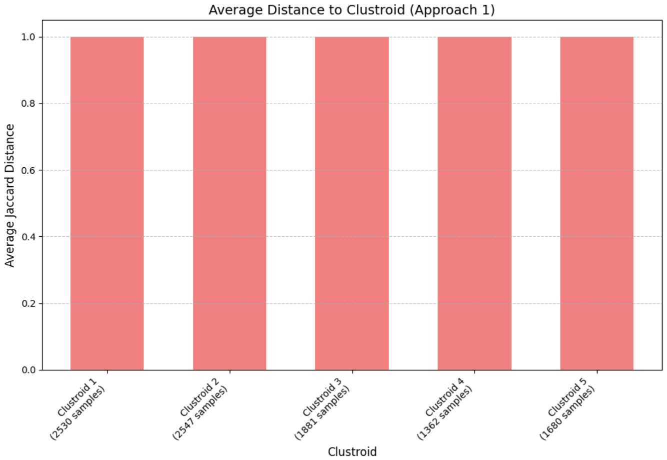

Fig. 1. Average Jaccard Distance to Clustroid

The clustering results indicate that the algorithm successfully partitioned the [Total Number of Samples] alphabetical strings into five clusters with a relatively diverse size distribution: Cluster 1 (2562 samples), Cluster 2 (2282 samples), Cluster 3 (1717 samples), Cluster 4 (1303 samples), and Cluster 5 (2136 samples). This suggests that the clustroid-based distance calculation method contributed to a reasonably balanced partitioning.

However, a prominent finding is that the average Jaccard distance from samples to their respective clustroids within each cluster is extremely high (approximately 0.999). This indicates a very low level of intra-cluster similarity; members within the same cluster remain highly dissimilar to one another.

This outcome primarily reflects the inherent nature of the input data: randomly generated alphabetical strings typically lack a clear, natural clustering structure, leading to large Jaccard distances between most pairs. Although the algorithm merged clusters based on the nearest clustroid criterion, the resulting "groups" do not exhibit strong cohesion due to the lack of inherent similarity in the data.

III. Second Task: Linear Regression – Gold price prediction

This task focused on predicting Vietnamese gold prices using a linear regression model implemented in PySpark. The objective was to transform historical time-series data into a suitable format for regression, train a model, and evaluate its predictive performance.

A. Overview of Linear Regression (LR)

LR is a supervised learning algorithm that models the linear relationship between a continuous target variable (y) and one or more independent predictor variables (features $\mathbf{x}$). The goal is to find an optimal linear function that best predicts y given $\mathbf{x}$.

For a single feature x, the model is:

$$y = \beta_0 + \beta_1 x + \epsilon \tag{1}$$

With multiple features $x_1, x_2, \ldots, x_p$, it extends to:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon \tag{2}$$

where β_0 is the intercept, β_j are the feature coefficients (weights), and ϵ is the error term. The coefficients are typically learned by minimizing a loss function, such as Mean SquaredError (MSE), often using optimization algorithms like L-BFGS. Key assumptions include linearity, independence of errors, and homoscedasticity. This project applies LR to predict gold prices based on historical price features.

B. Data Preparation

- **Dataset:** The primary data source was `gold_prices.csv` (2009/08/01 to 2025/01/01), read into a PySpark DataFrame.
- **Feature Engineering:** For each target date t, features were the respective ‘Buy Price’ or ‘Sell Price’ values from the 10 consecutive preceding days. PySpark’s `Window` functions and `lag` operation were used, followed by `VectorAssembler` to create feature vectors (e.g., `Previous Buy Price(s)`). 4000 samples were generated (`random_state=38`).
- **Data Splitting:** The generated DataFrame was randomly split into training (70%) and testing (30%) sets (`seed=2`).

C. Model Implementation and Training

Two separate Linear Regression models (`pyspark.ml.regression.LinearRegression`) were developed: one for ‘Buy Price’ and one for ‘Sell Price’, using their respective 10-day historical price vectors as features and the current price as the label. Models were configured with the `'l-bfgs'` solver and trained on the 70% training subset.

D. Experimental Results and Evaluation

1) Overall Results:

TABLE I
LR PERFORMANCE METRICS FOR GOLD PRICE PREDICTION.

Model	Data Set	RMSE	MSE	R²	MAE	Expl. Var.
Buy Price	Training	0.3232	0.1045	0.9995	0.1450	225.7249
	Test	0.2975	0.0885	0.9996	0.1356	226.5322
Sell Price	Training	0.3062	0.0938	0.9996	0.1406	240.1171
	Test	0.2913	0.0848	0.9996	0.1319	240.6501

The performance of the trained models was evaluated on both training and testing sets. Key metrics are summarized in Table I. The R^2 values (consistently $>$ 0.999), low RMSE/MAE, and high Explained Variance scores indicate strong predictive accuracy and good generalization to unseen data, with no significant overfitting observed.

2) Loss History During Training:

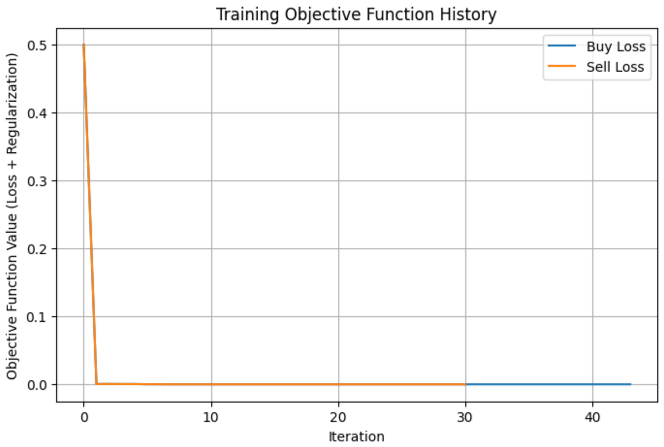

Fig. 2. Loss History of Buy Price Prediction Model

3) Performance Comparison:

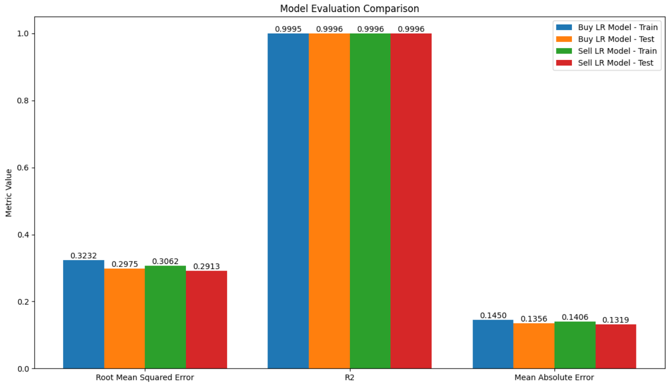

Fig. 3. 'Buy Price' and 'Sell Price' Models Performance.

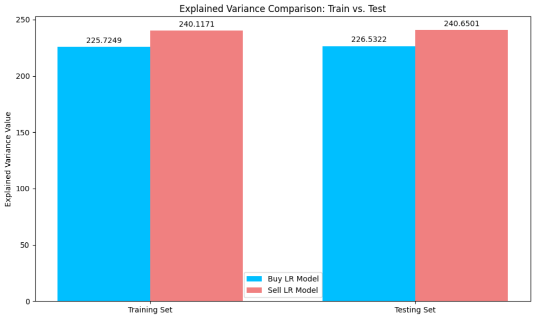

Fig. 4. 'Buy Price' and 'Sell Price' Models Explained Variance Performance.

IV. Third Task: CUR – Dimensionality Reduction

V. Fourth Task: PageRanking – the Google algorithm

A. Theoretical Basis

The project focuses on web crawling and PageRank computation to analyze the structure and importance of web pages within a specific domain. Web crawling systematically browses the internet to collect hyperlinks, forming a directed graph where nodes represent web pages and edges represent hyperlinks. The crawling process ensures that only valid URLs within the target domain are collected, excluding static files like images or PDFs to maintain relevance.

PageRank, developed by Google, is an algorithm that assigns importance scores to web pages based on their link structure. It models the behavior of a random surfer who navigates the web by following links with probability d (damping factor, typically 0.85) or jumps to a random page with probability $1 - d$. The PageRank score for a page p_i is computed iteratively using the formula:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{3}$$

where N is the total number of pages, $M(p_i)$ is the set of pages linking to p_i, $L(p_j)$ is the number of outgoing links from page p_j, and d is the damping factor. This iterative process converges when the difference between successive rank estimates falls below a tolerance threshold.

B. Implementation

The implementation consists of three main components: a web crawler to collect URL pairs, a PySpark-based framework for data processing, and a PageRank algorithm to compute page importance. The system is designed to handle large-scale data efficiently, leveraging parallel processing and distributed computing.

1) Crawler Function - Adjacency Matrix Initialization:

The web crawler is implemented in Python using the `requests` and `BeautifulSoup` libraries to fetch and parse web pages. The `crawl_website` function starts from a seed URL (`https://it.tdtu.edu.vn`) and collects unique (source, destination) URL pairs, representing directed edges in the web graph. It ensures that only URLs within the target domain are included and excludes static files (e.g., .jpg, .pdf) using the `is_valid_url` function. The crawler uses a set to store unique edges, avoiding duplicates, and saves the results to a CSV file (`url_pairs.csv`). The crawler initializes the adjacency matrix implicitly by collecting edges, which are later processed to form the graph structure for PageRank computation.

2) PySpark Initialization:

To handle large-scale data, the project uses PySpark, a distributed computing framework, to process the crawled URL pairs and compute PageRank scores. A Spark session is initialized to manage data as DataFrames, enabling scalable operations. The edge data from the CSV file is loaded into a Spark DataFrame with columns `source` and `destination`. The Spark configuration includes a checkpoint directory to manage intermediate results and prevent stack overflow during iterative computations. The edges are loaded and processed to create a list of unique pages (nodes) and compute out-degrees for each source page, forming the basis for the PageRank algorithm.

3) PageRank Implementation:

The PageRank algorithm is implemented in a `PageRank` class that takes the edge DataFrame, damping factor (0.85), maximum iterations (10), and tolerance (1e-4) as parameters. The algorithm initializes ranks uniformly ($\frac{1}{N}$) for all pages and iteratively updates them based on contributions from incoming links. It uses Spark's DataFrame operations for joins, aggregations, and broadcasting to optimize performance. Checkpoints are used every three iterations to manage memory. The algorithm outputs a DataFrame with page URLs and their corresponding PageRank scores, which are used for further analysis and visualization.

C. Result Analysis

The crawler collected 75,970 unique URL pairs from the `it.tdtu.edu.vn` domain, forming a directed graph of web pages. The PageRank algorithm was applied to this graph, and the top 20 pages by rank were visualized using a directed graph, with node sizes proportional to their ranks. The visualization was generated using the `networkx` and `matplotlib` libraries and saved as `top_pages_graph.png`.

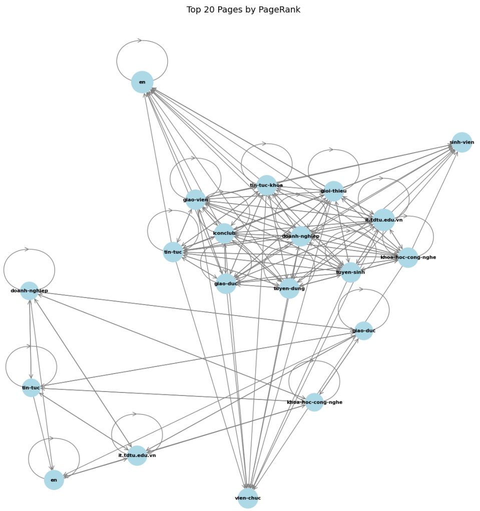

Fig. 5. Directed graph of the top 20 pages by PageRank score, with node sizes proportional to ranks.

The most important page, with a PageRank score of approximately 0.01293, was `https://it.tdtu.edu.vn/`, the faculty homepage. This high rank is expected, as the homepage is a central hub with many incoming links from other pages within the domain. Other high-ranking pages included department-specific pages and key sections like `https://it.tdtu.edu.vn/en`, reflecting the students' most focused topics and subjects.

VI. Contribution

The following table represents the contribution of each member, note that whichever member handles whichever task will also write the report for that task.

TABLE II
Member Contributions

ID	Member	Contribution	Progress
522H0036	Luong Canh Phong	Task 2 and Handling Report	100%
522H0092	Cao Nguyen Thai Thuan	Task 4 and Report Support	100%
522H0075	Tang Minh Thien An	Task 3	100%
522H0167	Truong Tri Phong	Task 1	100%

VII. Self-evaluation

The following table is our self-evaluation on our tasks:

TABLE III
Self-evaluation

Task	Task Requirements	Completion Ratio
Task 1	Hierarchical clustering in non-Euclidean spaces	90%
Task 2	Linear Regression – Gold price prediction	100%
Task 3	CUR – Dimensionality Reduction	90%
Task 4	PageRanking – the Google algorithm	100%
Task 5	Report	100%

VIII. Conclusion

References