Algorithms for Massive Data: Market-Basket Analysis

Thomas Mayr (12949A)

November 2024

1 Data Preprocessing

In this project, I analyzed the LinkedIn Jobs & Skills dataset to perform market-basket analysis, i.e. finding frequent itemsets, by implementing appropriate algorithms with a focus on scalability. To prepare the data for analysis, I undertook several preprocessing steps to ensure the dataset was clean, consistent, and suitable for frequent itemset mining.

I began by loading the dataset into a Spark DataFrame, selecting only the `job_skills` column for processing.

The first challenge I addressed was the presence of missing values in the `job_skills` column. Missing or null entries could lead to errors during processing and potentially skew the results. To mitigate this, I replaced any null values with empty strings. This ensured that all entries had a uniform format and could be processed without interruption.

Next, I transformed the `job_skills` strings into a more usable format. Each entry in the `job_skills` column originally contained a comma-separated string of skills. I split these strings into arrays, effectively converting each job's skills into a list of individual items. This step was necessary for treating each job's skills as a transaction in the context of market-basket analysis.

To standardize the data and ensure consistency, I trimmed any potential extra whitespace from each skill and converted all skill names to lowercase. This ensured that skills were uniformly represented, and should enhance the reliability of the frequency counts.

Additionally, I addressed the issue of potential duplicate skills within individual job postings. I accomplished this by converting each array of skills into a set and then back into a list, ensuring each skill appeared only once per transaction.

For testing and performance optimization, I decided to reduce the dataset size. I limited the analysis to a sample of 10% of the data. This mainly served to prevent the issue of running out of memory due to the limitations of the system used for the analysis.

Through these preprocessing steps, I transformed the raw `job_skills` data into a clean and structured format suitable for applying the Apriori algorithm.

2 Apriori Algorithm

The Apriori algorithm is a classic method used in data mining for frequent itemset mining and association rule learning over transactional databases. It operates on the principle that if an itemset is frequent, all of its subsets must also be frequent. This property, known as the Apriori principle, allows the algorithm to reduce the number of candidate itemsets by pruning those with infrequent subsets, thereby improving computational efficiency.

2.1 General Description of the Apriori Algorithm

The Apriori algorithm involves the following key steps:

1. **Initialization**: Start by identifying all frequent itemsets of size one (single items) that meet the minimum support threshold.
2. **Candidate Generation**: Use the frequent itemsets of size $k-1$ to generate a list of candidate itemsets of size k. This is typically done by joining the frequent itemsets with themselves and ensuring that all $k-1$ subsets of the candidates are frequent.
3. **Pruning**: Eliminate candidate itemsets that have any subset of size $k-1$ that is not frequent, based on the Apriori principle.
4. **Support Counting**: Scan the transaction database to count the support of each candidate itemset.
5. **Frequent Itemset Selection**: Retain the candidate itemsets whose support meets or exceeds the minimum support threshold. These itemsets are considered frequent.
6. **Iteration**: Repeat steps 2 to 5, incrementing k each time, until no more frequent itemsets are found.

2.2 Specific Implementation and Considerations

In this project, I implemented the Apriori algorithm using PySpark to handle the computational demands of processing the dataset efficiently. The primary goal was to identify frequent combinations of job skills listed in the `job_skills` column of the dataset.

2.2.1 Preparing the Transactions

I began by transforming the preprocessed DataFrame into an RDD (Resilient Distributed Dataset) of transactions, where each transaction represents the set of skills associated with a job posting. This conversion allowed parallel processing and manipulation using Spark's distributed computing capabilities.

2.2.2 Setting the Minimum Support Threshold

The minimum support threshold was defined as a fraction of the total number of transactions, denoted by `min_support_frac`. This approach allows for flexibility in adjusting the sensitivity of the algorithm to the dataset size. The final support threshold used in this analysis was 0.01.

2.2.3 Identifying Frequent Singletons

I initiated the algorithm by finding all frequent itemsets of size one:

- **Counting Item Frequencies**: I flattened the list of skills from all transactions and counted the occurrences of each skill.
- **Filtering Based on Support**: Skills that met or exceeded the minimum support count were considered frequent singletons.
- **Collecting Frequent Items**: I collected these frequent singletons for subsequent steps.

2.2.4 Filtering Transactions

To optimize the performance of the algorithm, I filtered each transaction to include only the frequent singletons identified in the previous step. This reduced the size of the transactions and focused the analysis on relevant items.

2.2.5 Iterative Process for Larger Itemsets

I entered a loop to find frequent itemsets of increasing sizes (from pairs onwards):

Candidate Generation

- **Joining Itemsets**: For itemsets of size k, I generated candidates by joining frequent itemsets of size $k - 1$ with each other.
- **Ensuring Correct Size**: I ensured that the union of itemsets resulted in candidates of the desired size k.
- **Pruning with the Apriori Principle**: Candidates with any infrequent subset were discarded. This was achieved by checking all $k - 1$ subsets of each candidate against the set of frequent itemsets from the previous iteration.

Support Counting

- **Broadcasting Candidates**: I broadcasted the list of candidates to all worker nodes to facilitate efficient counting.
- **Counting Occurrences**: I scanned the filtered transactions to count how many times each candidate itemset appeared.

- **Reducing and Aggregating**: The counts were aggregated to determine the total support for each candidate.

Selecting Frequent Itemsets

- **Filtering Based on Support**: Candidates meeting the minimum support threshold were retained as frequent itemsets.
- **Updating the List of Frequent Itemsets**: I collected these itemsets and extended the final list of frequent itemsets.

Iteration Control

- **Termination Condition**: If no new frequent itemsets were found in an iteration, the loop terminated.
- **Incrementing k**: I incremented the size k for the next iteration to find larger itemsets.

2.2.6 Memory Management and Performance Considerations

- **Data Structures**: Throughout the implementation, I used `frozenset` to represent itemsets. `frozenset` is an immutable and hashable data structure, allowing us to store itemsets in sets and use them as dictionary keys efficiently. Normal sets in python are muteable and can therefore not be used as dictionary keys.
- **Encoding Skills to Integers**: I extracted all unique skills from the dataset and created a mapping from skills to integers. By using the integer representation in the algorithm and only afterwards decoding the results for interpretability, I improve computational effiency and memory management.
- **Broadcast Variables**: I used broadcast variables to distribute the candidate itemsets to all worker nodes. Broadcasting candidates ensures that each worker node has a local copy, avoiding repeated data transfers. After each iteration, I unpersisted the broadcast variables to free up memory.
- **Collecting Data to the Driver**: I minimized the data collected to the driver node to avoid memory bottlenecks, especially when working with larger datasets.
- **Optimizing Candidate Generation**: By pruning infrequent candidates early, I reduced the computational load in subsequent iterations.

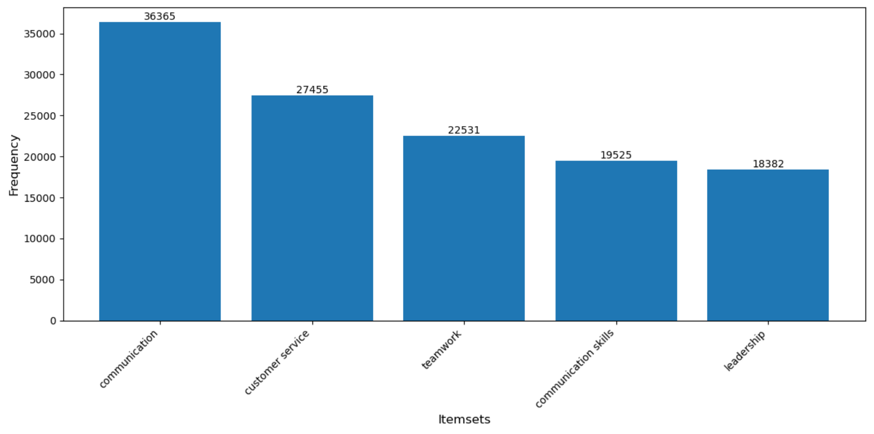

Figure 1: Top 5 Frequent Itemsets of Size 1

2.2.7 Final Results

After completing the iterations, I obtained a list of frequent itemsets along with their support counts. These itemsets represent combinations of skills that frequently occur together in job postings. As we can see, the most frequent itemsets of all sizes unsurprisingly consist of somewhat generic soft skills like ”communication”, ”teamwork” or ”leadership”.

3 Discussion

3.1 Scalability

One of the primary challenges of the Apriori algorithm is its computational complexity. The algorithm generates a vast number of candidate itemsets, especially when dealing with datasets that have a large number of frequent items. As the size of the itemsets increases, the number of possible combinations can grow exponentially. This leads to significant consumption of computational resources in terms of both time and memory.

Indeed in the testing phase of this project, it was found that using a larger sample of the data with a lower support threshold often lead to either significantly more time consuming computations or even complete failure due to running out of main memory. However, on a machine with a memory of 16GB and a support parameter set to 0.01 the algorithm managed to successfully find all frequent itemsets within the full dataset without failing.

While we might be able to mitigate the problems of the Aprior algorithm by carefully managing memory and optimizing candidate generation or simply by counting upon more capable hardware, a more reliable solution would be to use alternative algorithms that have reduced computational overhead and improved

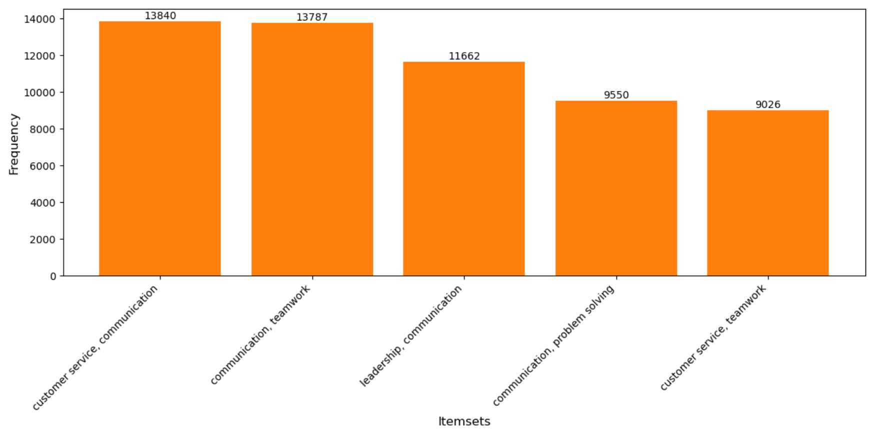

Figure 2: Top 5 Frequent Itemsets of Size 2

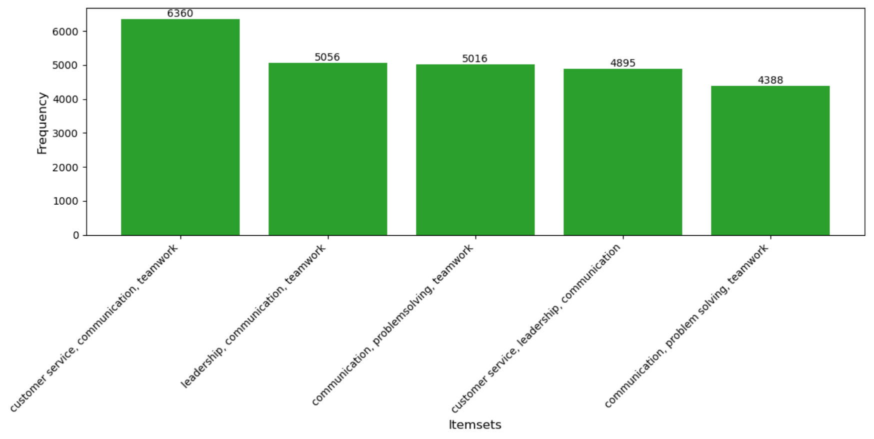

Figure 3: Top 5 Frequent Itemsets of Size 3

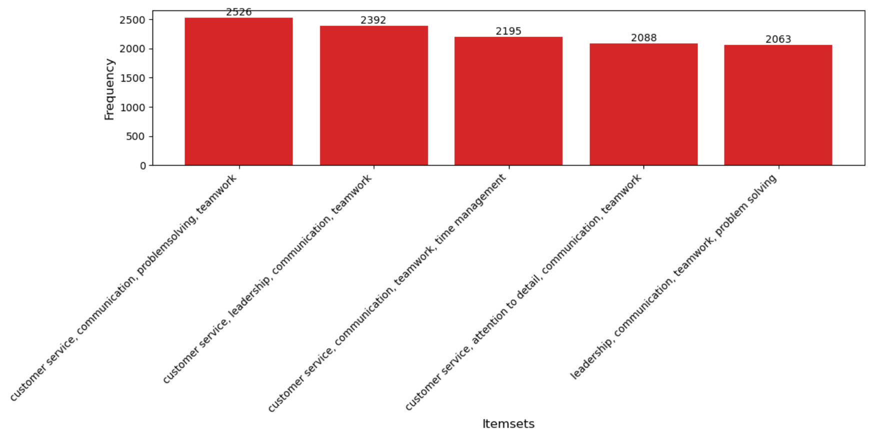

Figure 4: Top 5 Frequent Itemsets of Size 4

efficiency.

When I tested out a popular alternative to Apriori, the FP-Growth algorithm, I found that it was able to handle larger sample sizes and lower support thresholds without running out of memory and generally performed better in terms of computation time. The FP-growth algorithm avoids the candidate generation step and works with a compact data structure called the FP-tree. Note that the FP-growth algorithm was not implemented from scratch in this project and was only used for this simple test comparison due to its popularity and built-in availability in PySpark.

While the Apriori algorithm has somewhat limited capabilities, especially on systems with computational ressources not intended for massive data analysis, it performs reasonably well on many datasets, especially when used in combination with Spark, RDDs and the MapReduce framework, like in this implementation, and moreover has the added benefits of being relatively easy to implement and adapt.

3.2 Possible Applications

When we look at the results of our analysis what we can immediately notice is that the most frequent itemsets we find are often composed of extremely generic skills like ”teamwork”, ”communication” and so on. While we successfully implemented the Apriori algorithm and demonstrated its capabilities and limitations, the useability of the results themselves is quite limited in this form.

An interesting extension of this analysis might be to create an application that lets the user search for a specific skill to see if it's in high demand, i.e. if it is a frequent item. Going one step further, we could integrate association rule mining in our analysis and allow users to input a specific skill or a set of skills and receive recommendations for additional skills that are commonly associated with their input. For example, a user proficient in ”data analysis” might discover that this skill is often associated with ”machine learning” and ”python”. Recognizing these associations could encourage the user to pursue learning opportunities in these areas, thereby enhancing their employability and alignment with industry demands.

4 Conclusion

I successfully implemented the Apriori algorithm using PySpark and Resilient Distributed Datasets in combination with MapReduce and demonstrated its capabilities on a LinkedIn dataset consisting of skills demanded in job postings. The algorithm was able to find frequent itemsets within a reasonable time frame, but failed when I chose a low support threshold and a high sample size due to running out of main memory. The issue persisted even when I undertook several optimization steps. In a professional setting, where availability of computational ressources is not an issue, the Apriori algorithm continues to be an attractive option due to its relative simplicity and adaptability. Although, for more demanding computations more efficient algorithms for frequent itemset mining should be considered. I also briefly discussed possible applications of this specific analysis and considered a skill recommendation system as a potentially meaningful extension of a project like this.