university of groningen

Scalable Computing

Spring 2019

Group ID - 41

Michael Yuja (s3766365)

Ankit Mittal (s3683974)

Carlos Huerta (s3743071)

Project Description

The objective of this project is to develop a product recommendation engine for an ecommerce website, www.agglobal.com. The website has above 7000 products listed, and it can be hard for users to navigate around all the products in order to find what they need. We aim to implement a system based on the company's historical transaction data to add a feature similar to Amazon's "People who bought x also bought y …". When viewing a specific product, our system should be able to recommend a list of products that would complement the product that the user is viewing. Similarly, when a user adds products to his/her cart, our system should recommend other products based on what is currently in the cart.

The firm has large volumes of transactional sales data that span over 4 years. Due to the size and nature of this data, parallelization and scalability needs to be implemented in order to process it and create recommendations that could serve our clients with new and exciting products. The engine will mine association rules from this dataset and update them as the company generates more data over time.

Algorithm - FP Growth

The end goal of our engine is for it to be able to find items that are frequently bought together. The firm has provided us with transaction data in which each transaction contains the unique item codes of products that were bought together. Our current dataset contains around 700,000 transactions.

There are several algorithms and methods that are used to make recommendations. The Apriori algorithm is notably one of the most famous, but it is also very slow on huge amounts of data. The FP Growth algorithm, introduced 19 years ago by Han et al, is an order of magnitude faster than apriori without parallelization. The vanilla Fp-Growth algorithm is constructed in a recursive way, we first compute a list of frequent itemsets and then we use this constructed frequent item list to sort the original transactional data removing non-frequent patterns. Finally we can recursively construct the fp-tree with each processed transaction. The fp-tree is used alongside a header table that contains references to each node on the constructured tree to extract transactional patterns from each node that contains frequency information of this patterns. Then these frequencies are used to calculate the support and confidence of the proposed relevant rules of our data.

This algorithm has various scaling challenges if implemented on a single machine, first the size of the generated tree can be an issue if the size of your transactional data is huge, also the process of traversing the tree, also done in a recursive way, is also very computationally expensive. This makes the traditional fp-growth algorithm not very scalable friendly.

Algorithm - PFP Growth

The parallel fp-growth algorithm proposed by Li et al in 2008 uses a divide and conquer approach. The steps we took for implementing it are detailed

1. First we compute the the list of frequent itemsets as on the original fp-growth algorithm and sort them by frequency in descending order. This process will discover the items vocabulary **I,**
2. The next stage is to group the items dividing all the items in **I** into **Q** groups. The list of the groups will be called the Group-list, each group is given an unique identifier (groupid) that will be used as a key on the map phase of our mapreduce algorithm.
3. Then the mapping phase begins, using these group ids we output one or more key-value pairs per transaction. The value being generated is a group dependant transaction.
4. After that, in the reduce phase, we group by groupid and compute the conditional databases on the grouped transactions, and then mine each of the individual conditional fp-trees in the normal way. Then we calculate the confidence of each of the proposed rules.
5. The final step is collecting all the locally produced rules and save them into a database to be used as a lookup table for production with the product codes as id's.

We opted for this algorithm because it can indeed be parallelized and doing so will result in very fast mining. This will permit us to run the mining much more often and be able to include new data more frequently.

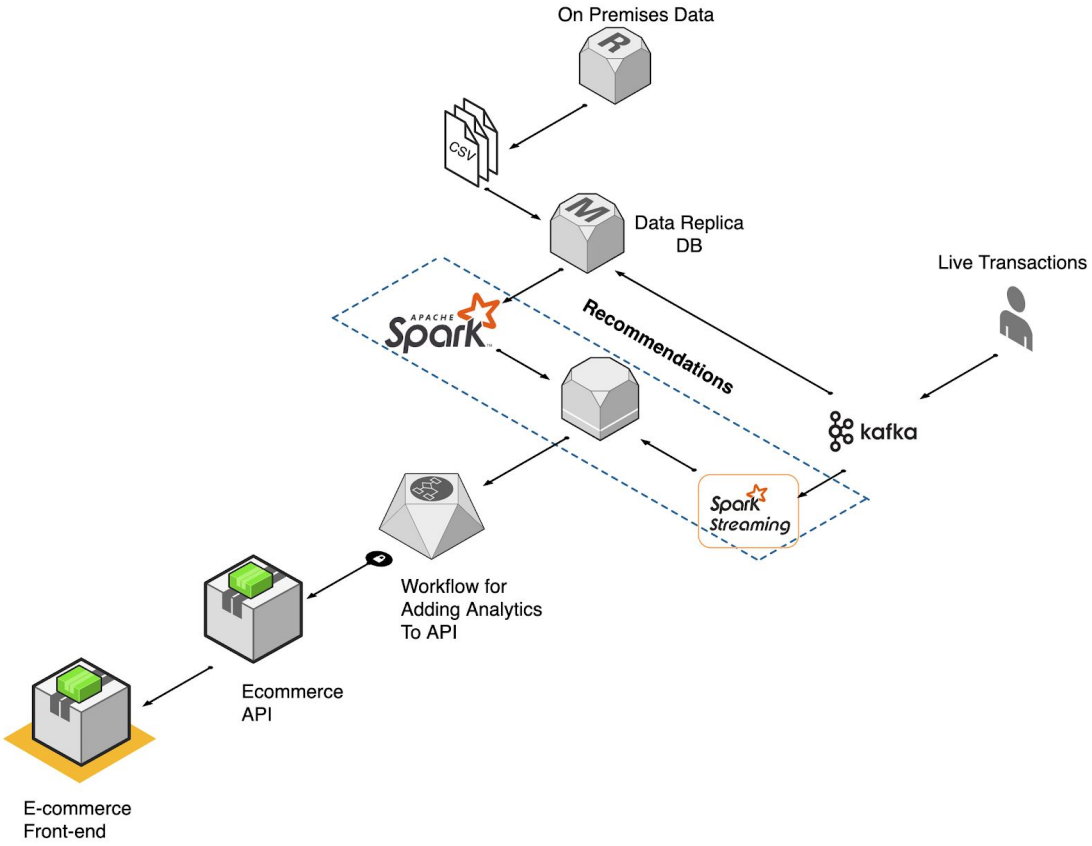
The output of the algorithm is a table, as shown below in Table 1:

Bought Item (antecedent)	Suggested Item (consequent)	confidence
[023005]	[015350]	0.439
[035858]	[035840]	0.427

The antecedent is a list of items for which a recommendation should be generated. The consequent is a list of items that are recommended to the user given the antecedent. The confidence is the level of certainty that a user who buys the antecedent will also buy the consequent. The recommendations can then be stored in the company database as a lookup table and easily connected to the web API.

Architecture

Our project is an add on to an existing system belonging to agglobal.com. As such, we have designed it in such a way that it allows us to integrate with it with minimal dependency on the company's systems due to both practical and security reasons. The database admin at the company exported the batch data for us as CSV, so we take this as our initial input. We use the CSV to create a local replica DB of the data. This is shown in the diagram below:



Using this DB as a source, we pull the data into Spark to begin the processing. The output is the lookup table we have previously described, which is then transferred back to the company's servers in a CSV format. The company engineers are then in charge of developing the API endpoint that will handle passing the list of items to lookup recommendations for, and receive the recommended items' codes.

Technology Stack

Our current technology stack is the following:

1. Database - MongoDB for the Data Replica
2. MapReduce - PySpark
3. Message Queue - Kafka
4. Processing Data Streams - Spark Streaming
5. Containerization: Docker (Using docker-compose for stack deployment)

Design Choices

Since the start, we set out with the objective of making our solution scalable. The technologies we chose for both storage and message queuing - MongoDB and Kafka - are built to have fault-tolerant, distributed storage of data. Spark is then capable of processing data in a distributed manner. It can assign *N* number of workers to take care of any given job. We then built our FP Growth using mapReduce so that it is able to parallelize the different steps in across all the workers that are assigned to the job. Finally, our cluster is completely dockerized, so deploying it in a cloud service will be relatively easy to do so, and to provision more resources as well.

Insights

The main function of our algorithm is to produce reasonable recommendations in a sub-10-minute time period. In this regard, we believe our project to be successful. This is one example that was generated by the algorithm (2nd row in Table 1):

Antecedent	Consequent
	

--------->

VOLTECK.
PLACA PARA TOMA DOBLE VOLTECK
PLASTICA 46406

Código: 035858
Clave: PPDO-S
Disponibilidad: ✓

VOLTECK.
TOMA DOBLE VOLTECK SIN PLACA
15A 125V POLAR 46102

Código: 035840
Clave: CODO-S
Disponibilidad: ✓

By only looking at the pictures of what the products are, we can already tell that this successfully recommends a product that is very relevant to the antecedent. In the database, although these products have codes that are similar, there is nothing that relates them to each other (besides the brand.) We feel that this algorithm has given the application a way to relate products that otherwise had no connection.

Although the confidence levels that resulted from some of our data seem low, they are enough for the purposes of our algorithm. The cost of recommending a product which is less likely to be bought based on a certain confidence threshold is not high - we would rather recommend products with low confidence values than not recommend anything at all.

For the streaming part of our project, we thought that it didn't make much sense to use streaming data to update the association rules. If we get i.e. 100 new transactions in a 10 minute

window, then we would have to run the FP Growth algorithm on the entire data set with the new transactions added to it, as the running the algorithm only on the stream data won't be useful because it won't have any effect on the generated association rules . In this case, it would be better to just trigger batch processes after a certain period instead of using stream data as it doesn't take much time to run the algorithm on the whole dataset. Instead, we decided to stream new transaction data and count the number of times that a certain product has been bought in a given window. Using this count, we can find the top 5 most bought items during this window and display those in the home page of the website.

We agree, however, that a cloud-based implementation would be more suitable for a production environment. In this way, we can achieve communication with the site's API more easily. Given more time, we would have uploaded our application into a cloud based environment that supports container orchestration, our project is ready to be deployed into the cloud as the containerization work that has been done from the beginning makes this a natural way to proceed.