Task 3 - Model Development, Final Report

Group 10

Topic: Scalable and Efficient Product Search Architecture.

Team Members: Adit Shrimal, Manav Middha, Neil Chitre, Varun Hande

Motivation: The online retail industry has grown tremendously over the years, leading to a large volume of product data being generated. An efficient and scalable product search solution is crucial in providing a seamless shopping experience to customers. This project aims to address the issue of building a scalable and efficient product search architecture that can handle large amounts of data and provide accurate results to users.

Data sources (URL): Asos Product real-time APIs (Rapid API).

List of products: https://asos2.p.rapidapi.com/products/v2/list
List of Categories: https://asos2.p.rapidapi.com/categories/list

ML Objectives: We experiment with the following different techniques to get:

1. Use TF-IDF in SparkML to create word vectors and calculate cosine similarity to get similar products.
2. Use Word2Vec in SparkML to generate word vectors and calculate cosine similarity to get similar products.
3. Implement BM25 in Spark for product search in constant time.
4. Use BERT to create word vectors and calculate cosine similarity to get similar products.

Overview of Data Engineering Pipeline:

Our Data Engineering pipeline is divided into Airflow parts:

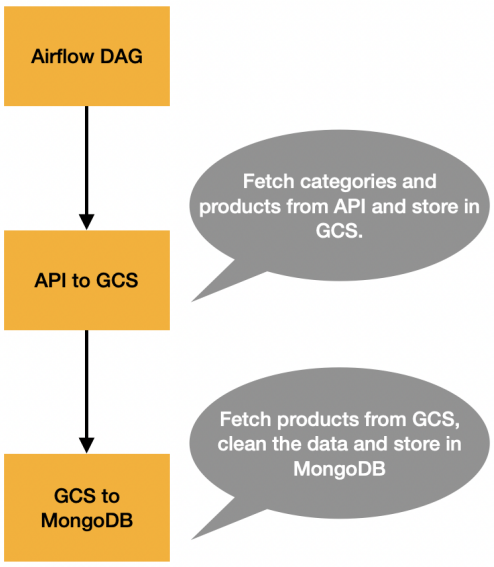
Fetch data from API to GCS:

1. Fetch the categories data from the following API:
    a. List of Categories: https://asos2.p.rapidapi.com/categories/list
    b. Store the categories in GCS and MongoDB.
2. For each category, we fetch the products from the following API:
    a. List of products: https://asos2.p.rapidapi.com/products/v2/list/{category_id}
    b. Store the products in the GCS.

GCS to MongoDB:

1. Fetch the products from GCS.
2. Process the raw product data, keep the required fields and store each product as a document in MongoDB under the 'products' collection.

Pipeline Overview:



Cluster Specification on GPU:
Databricks Runtime Version - 12.1 ML (includes Apache Spark 3.3.1, GPU, Scala 2.12).
Worker/Driver types - g4dn.xlarge
Number of workers - min 2 | max 5
2-5 workers - 32-80 GB Memory8-20 Cores
1 Driver - 16 GB Memory, 4 Cores

Cluster Specification on CPU:

2-5 Workers
61-152.5 GB Memory
8-20 Cores
1 Driver
30.5 GB Memory, 4 Cores
Runtime
7.3.x-scala2.12

Preprocessing Goals and Algorithms:
We read the product data from Mongo Atlas. Our database currently contains ~26K products. Since the product names are not clean, we tokenise the data and remove stop words from getting a better form of data to be passed for developing word vectors.

Input Data:

```
+---------+-----------+-------------------------------------------------------------+
|_id      |brandname  |name                                                         |
+---------+-----------+-------------------------------------------------------------+
|204234534|A.Kjaerbede|A.Kjaerbede Kaya aviator sunglasses in smoke transparent     |
|204234523|A.Kjaerbede|A.Kjaerbede Fame square sunglasses in black                  |
|204234586|A.Kjaerbede|A.Kjaerbede Jake flat top round sunglasses in gray transparent|
|204234620|A.Kjaerbede|A.Kjaerbede Bror sunglasses in havana                        |
|204234621|A.Kjaerbede|A.Kjaerbede Fame square sunglasses in burgundy transparent   |
+---------+-----------+-------------------------------------------------------------+
only showing top 5 rows
```

Output Data:

```
+---------+-----------+-----------------------------------------------------------+-----------------------------------------------------------------+
|_id      |brandname  |name                                                       |filtered_words                                                   |
+---------+-----------+-----------------------------------------------------------+-----------------------------------------------------------------+
|204234534|A.Kjaerbede|A.Kjaerbede Kaya aviator sunglasses in smoke transparent   |[kjaerbede, kaya, aviator, sunglasses, smoke, transparent]       |
|204234523|A.Kjaerbede|A.Kjaerbede Fame square sunglasses in black                |[kjaerbede, fame, square, sunglasses, black]                     |
|204234586|A.Kjaerbede|A.Kjaerbede Jake flat top round sunglasses in gray transparent|[kjaerbede, jake, flat, top, round, sunglasses, gray, transparent]|
|204234620|A.Kjaerbede|A.Kjaerbede Bror sunglasses in havana                      |[kjaerbede, bror, sunglasses, havana]                            |
|204234621|A.Kjaerbede|A.Kjaerbede Fame square sunglasses in burgundy transparent |[kjaerbede, fame, square, sunglasses, burgundy, transparent]     |
+---------+-----------+-----------------------------------------------------------+-----------------------------------------------------------------+
only showing top 5 rows
```

For doing the above, we use RegexTokenizer and StopWordsRemover from SparkML.

RegexTokenizer: Helps to filter the data according to a given regular expression.
Reference: https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.ml.feature.RegexTokenizer.html

StopWordsRemover: Filters out common words like a, an, and the from the data which do not have any impact in describing the data.
Reference:
https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StopWordsRemover.html

Time Taken for preprocessing: 2.5 seconds on GPU and ~ 1 second on CPU

ML Goals: Our main objective in this project is to use various NLP techniques to find contextual similarity between the product being searched and the corpus of products to show the most similar products.

We experiment with the following methodologies:

1. TF-IDF + Cosine Similarity: Term frequency-inverse document frequency (TF-IDF) is a feature vectorisation method widely used in text mining to reflect the importance of a term to a document in the corpus. Once the

vectors are created for the search query and the data corpus, we use cosine similarity to find the most similar products for the search query.

Reference: https://spark.apache.org/docs/latest/mllib-feature-extraction.html#tf-idf

2. Word2Vec + Cosine Similarity: Word2vec tool processes a text corpus to generate word vectors as output. The tool first builds a vocabulary from the training text data and then learns the vector representation of words. The vectors for the search query and data corpus are created, and cosine similarity is utilised to identify the most similar products for the search query.

   Reference: https://spark.apache.org/docs/latest/mllib-feature-extraction.html#word2vec , https://code.google.com/archive/p/word2vec/

3. BM25: BM25 is a ranking function that evaluates the relevance of documents based on query terms and their frequency in the documents. It uses term frequency and inverse document frequency for calculating the BM25 score.

   Reference: https://docs.vespa.ai/en/reference/bm25.html

4. BERT: Bert is a pre-trained model, trained on huge corpus data, which maps sentences to a high dimensional space which can then be used for product similarity using cosine similarity. We use the all-MiniLM-L6-v2, transforming our data into a 384 dim vector

   Reference: https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-cos-v1

Outcomes: Out of all the models that we developed, TFIDF is the easiest to interpret but has its limitations in being able to capture the semantic meaning of data. Word2Vec can only handle search queries that are present in the corpus on which the model is trained and ends up giving erroneous results. BM25 is very quick but needs to give results based on semantic meaning. Our best model is BERT, as it can understand the semantic meaning of data very well and fill in the gaps that our above models need to include.

Execution Time for Search:

S.No	Model	Average Execution Time on GPU	Average Execution Time on CPU
1.	TF-IDF + Cosine Similarity	7 seconds	4 seconds
2.	Word2Vec + Cosine Similarity	10 seconds	7 seconds
3.	BM25	0.005 seconds	0.005 seconds
4.	BERT + Cosine Similarity	0.07 seconds	0.07 seconds

Execution Time for Model Training:

S.No	Model	Average Execution Time on GPU	Average Execution Time on CPU
1.	TF-IDF + Cosine Similarity	5 seconds	2.2 seconds
2.	Word2Vec + Cosine Similarity	8.76 seconds	5.05 seconds
3.	BM25	1 minute for inverted indexing	1 minute for inverted indexing
4.	BERT + Cosine Similarity	16.82 seconds	1.65 minutes

Search using different algorithms perform faster on CPU for all algorithms as compared to GPU. We observed that model training is also faster for all algorithms on the CPU except for BERT, which is faster on GPU.

Lessons Learned and Next Steps: In this project, we learned to implement NLP techniques like Word2Vec and TFIDF in a distributed fashion using PySpark libraries. We also implemented BM25 in the spark from scratch.

Since we observed that BERT performs the best, we plan to research and implement BERT in a distributed fashion using SparkNLP. This research should help us in getting similar results in a faster way.

Conclusion: From the execution time and meaningfulness of the search results, we find that BERT is the best model since it is built upon a large corpus of data and can handle search queries in multiple languages. BM25 is the fastest method and is the most scalable.