Problem Statement

Given a search term and the articles discovered by Blackwing for that term, sort the articles by relevancy.

The MLDS pipeline will operate AFTER articles have been found, meaning we will not be searching all possible articles for the most relevant.  Our results will be restricted to what Blackwing has already identified as relevant, regardless of whether or not they are accurate results.

This will be easier for us to implement from an engineering perspective than if we had to search all articles that exist.  It also eliminates the need to overhaul Blackwing's internal search capability.

Architecture

The MLDS pipeline has several components, or modules.

Data Ingestion (Optional Step)

At this time, we are reading the article data from the MySQL database and performing no cleaning or pre-processing of any kind.  We could add this step in the future, if needed.

Syndicated Content Part 1: Exact Duplicates

The MD5 hash is a fast and simple way to identify duplicate articles.  It hashes the article text.  In this step, 1 sample document from each group of identical articles is selected for further processing.  Since a group contains identical text, it makes no sense to process the same text more than once.  It does not matter which document from a group is chosen, because the texts are all the same.  The article ID (search\_hit\_id) is stored so that it can later be linked back to the group and so that all identical articles will be displayed in the search results.

Identical articles will have the same relevancy scores and be grouped together in the search results.

Embedding

An embedding is a numerical representation of text.  There are many embedding model options.  We will use one called the Universal Sentence Encoder (USE).

**Advantages of USE Over Other Embedding Models**

It was trained specifically for the task of finding semantically similar content.  Its training data consisted of semantically similar sentence pairs.

When comparing embeddings, even a small difference in a single dimension can have an impact on similarity or distance between embeddings.  USE learns embeddings that have a wider range of values in the embedding dimensions, allowing it to better separate close matches in the embedding space (interquartile range of 0.5 - 0.8 VS 0.79 - 0.87 for BERT embedding models).

**Limitations**

USE can be used for a variety of downstream similarity tasks, like semantic search, semantic similarity, and question answering.  However, **USE should NOT be used for other tasks, like text summarization, sentiment, etc**.

At the current time, we are using USE to create the embeddings for the ad prediction model.  This is not optimal, but we are doing this to simplify development so that we can meet deadlines.  In the future, it would be better to use a different type of embedding for ad prediction.

Feature Store Creation with FAISS Index

Computing the similarity between a set of article embeddings requires pairwise comparisons, with a complexity of O(n^2).  The computation time explodes when the number of articles becomes large.  Facebook's FAISS package was designed to reduce computation time for comparing vectors in high dimensional space.  It uses several algorithmic tricks to accomplish this, but the bottom line is that it allows us to more efficiently calculate the distances between embeddings.  More distant embeddings are less similar, thus, FAISS enables both semantic similarity and semantic search.

A FAISS index is a feature store of sorts, because it can efficiently store a low dimensional representation of higher dimensional data such that it can be easily reproduced.  The embeddings can be re-constructed using the FAISS index.  FAISS uses a locality sensitive hashing (LSH) method under the hood, just like SimHash.  Our pipeline will save the FAISS index for the articles ingested.  This index can be fed to many downstream tasks, or used as part of Haystack's many pipelines.

In the future, if we wanted to add a question answering capability, for example, we could layer a Haystack retriever on top of the FAISS document store, and it would be ready to go.

Syndicated Content Part 2: Near Duplicates (Optional Step)

The pipeline reduces data to be processed by subsequent steps through the identification of text similarity.  FAISS can be used to identify semantic similarity, which could reduce the number of documents that would need to pass through the semantic search module.

However, there is the question of whether near duplicate, or non-exact match syndicated content should be identified at all.  If our ultimate goal is to return relevant results for the search term, it might not make sense to group search results by similarity when there is the possibility of it changing the search result order.  See the example below.

If we care more about relevancy to the search term than grouping semantically similar search results, then we do not need a module to identify near duplicates.  That being said, the pipeline could be used to identify non-exact match syndicated content for other Cision products.

Semantic Search

FAISS' range search returns distances between a query vector and embedded articles.  It can be considered a proto-relevancy score.  This module scores every article's semantic similarity to the search term.

Ad Prediction

Advertisements, or articles containing advertisements, should be ranked lower in the search results.  So this module includes a custom ad prediction model that was trained on open source data.  The model's ad probability scores will factor into the final relevancy score.

The ad prediction model takes the article embeddings as input.  Therefore, it can run in parallel with the FAISS index creation, near duplicate detection, and semantic search modules.

Relevancy Scoring

The relevancy score determines the order by which search results will be sorted.  This module combines the ad probability scores with the semantic search distances for a final relevancy score.

How FAISS Works

FAISS similarity search is central to this pipeline.  This section describes the technical details behind it.

The Problem it Solves and the Algorithmic Tricks it Uses

If you need to calculate the distance (or similarity) between every 2 pairs of high dimensional vectors, the time complexity for a purely pairwise comparison is O(n^2), which blows up as the number of vectors, n, increases.  FAISS reduces the time complexity by using algorithmic tricks.  The first trick is to use an Inverted File Index (IVF Index).  The second trick is to approximate the L2 (Euclidean) distance between the vectors, rather than calculating it directly, by using Product Quantization (PQ).  Of the two tricks, the second saves the most time by trading accuracy for speed.  The Python FAISS library offers functions for each trick, independently, so if you only want to implement the IVF Index and not PQ, you can.  It also offers a FlatL2 index, which is the baseline of a pairwise comparison with L2 distances.

**Inverted File Index (IVF Index)**

Imagine you are the mayor of a town and a new person arrives in town looking for a place to live.  This new person has specific likes and dislikes.  You can think of the person's preferences as their feature vector.  Everyone who lives in your town has their own likes/dislikes too.  If you wanted to match this newcomer with the best home, you could compare the newcomer's feature vector to the feature vector of everyone who lives in your town.  This would be a pairwise comparison.  But you are not just a mayor, you are a smart data scientist, and you know that there is a more efficient way to do the comparison.  You divide your town into districts, because you know that the townsfolk with similar likes/dislikes tend to live near one another.  You ask for 1 representative for each district to come forth and compare their feature vector to the newcomer's.  Now you no longer have to do a pairwise comparison with everybody, only the smaller group of district representatives.  That is the logic behind the IVF Index.

So how do you create the districts?  One option is Locality Sensitive Hashing (LSH).  LSH hashes vectors such that vectors that are more similar end up with hashes that are closer together.  Like any hash function, the same input produces the same hash every time.  Unlike cryptographic hash functions however, LSH actually tries to create collisions, because it wants to group hashes into buckets that contain similar hashes.  FAISS offers a Flat LSH Index function.  The drawback to LSH is that it has a high memory cost.

Instead of LSH, another option for creating the districts is to use a dirichlet tessellation (a.k.a. Voroni cells).  That is a fancy way of saying you are going to scatter marbles over a map of your town and treat each marble as a centroid.  Then you will expand the space around each centroid until they collide, forming the district boundaries.  This is what an IVF Index does.

When you create a dirichlet tessellation like this, you need to decide how many centroids you want (how many marbles are you going to roll?).  That is a hyperparameter of the IVF Index called "nlist".  The more centroids you have, the more accurate your search will be, but it will take longer.  The fewer centroids you have, the less accurate your search will be, but it will be faster.

By searching this way, you only need to compare the newcomer's feature vector to the centroids.  Once you have found the closest centroid, then you can do a pairwise comparison with the feature vectors in that centroid.

A reasonable question would be: what if your query vector (your newcomer) has a feature vector that places it along the boundary of a Voroni cell?  The newcomer might be closer to a particular cell's centroid, but it might be more similar to a vector just over the boundary.  This is where another hyperparameter called "nprobe" comes into play.  nprobe determines how many neighboring Voroni cells you will consider in your search.  If your newcomer is allowed to be compared to a few neighboring cells, they might find a better vector to call similar (a better home to live in).  Just like nlist, the larger nprobe's value, the more accurate the search will be, but it will take longer.  A lower nprobe results in a less accurate search, but it will be faster.

**Product Quantization (PQ Index)**