# Semantic Vector Search for Twitter Data with HNSW Retrieval

By Justin Vita, Jonah Bishop, Lax Kyada, Nicholas Raines, Samarth Mehta

## 1 Background/Introduction

Computers and humans understand language very differently. To humans, words come together to create meaningful sentences that convey information but to computers sentences and documents are simply collections of words that can be viewed as tokens and understood only in their context relative to other documents. This fundamental disconnect between computers and their users can make information retrieval systems faulty because two phrases that have the same semantic meaning could be unrecognizable to a computer if they do not contain similar tokens.

This is the main shortcoming of keyword search, the solution to searching using natural language with computers. The most basic example of this is ‘ctrl+f’ on a webpage. You type in a phrase and the computer finds for you any areas on the webpage that that exact query is matched. A slightly more complex system might take in your input, break it into words, and return any sentence or paragraph that contains most or all of the words that you typed but maybe not in order. An even more complex system would add in functionality like stopword removal; where words that likely don’t have much impact on the meaning of the query are removed to highlight the words that, through their uniqueness, shine more light on the intent of the search, or stemming which normalizes the tense of each word to cluster words that have the same basic meaning together Recent developments in natural language processing have produced language models that are capable of encoding the semantic meaning into vector space using machine learning. Our project is an application of this for an example use case, Twitter.

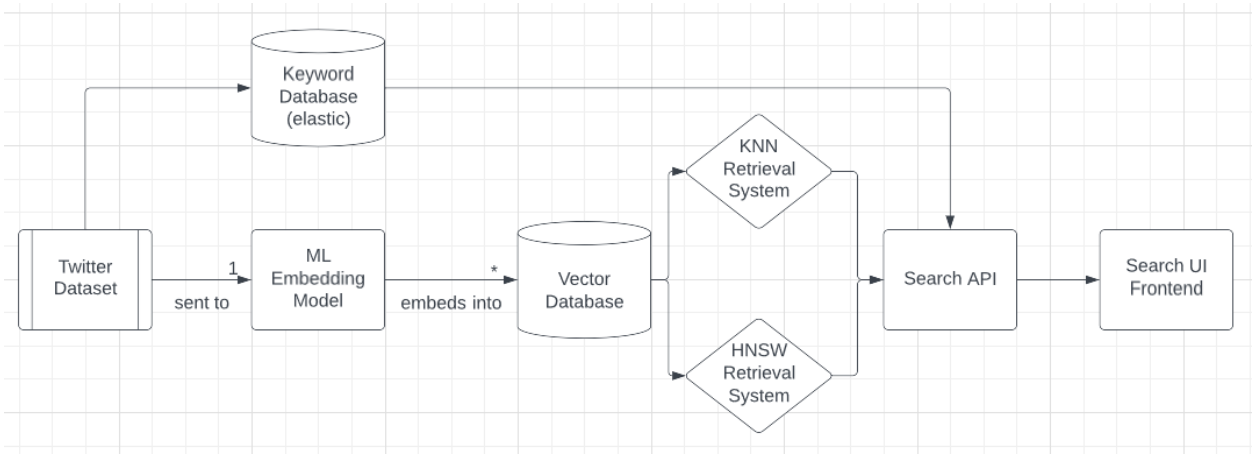
The reason we chose Twitter as our use case is that Twitter’s documents are very short with a 280 character maximum, and users usually only interact with an individual tweet for a few seconds. This means that their queries have a harder time finding the documents due to their small size impacting the effectiveness of keyword searching and that users will often not remember the exact phrasing of a tweet when searching for it. We wanted to explore vector search as a possible alternative.

## 2 Project Description

The system that we wanted to develop is a multimodal vector search system with an HNSW backend. Unfortunately due to the computing power and time that it would take

to embed and store a dataset that includes images we weren't able to test their inclusion, although our system should theoretically fully support it.
One design decision that we made early on in our project was that since we aren't able to integrate our system with Twitter in any meaningful way we would instead make our project a comparative analysis and proof of concept for a vector search system versus a keyword search system or Twitter data. This means that in addition to the components necessary for the HNSW vector search system we also needed to implement a KNN vector search system to see the effect of an ANN backend as well as an elasticsearch keyword search retrieval system to see the improvements and drawbacks of a vector search system.

### 2.1 System Design



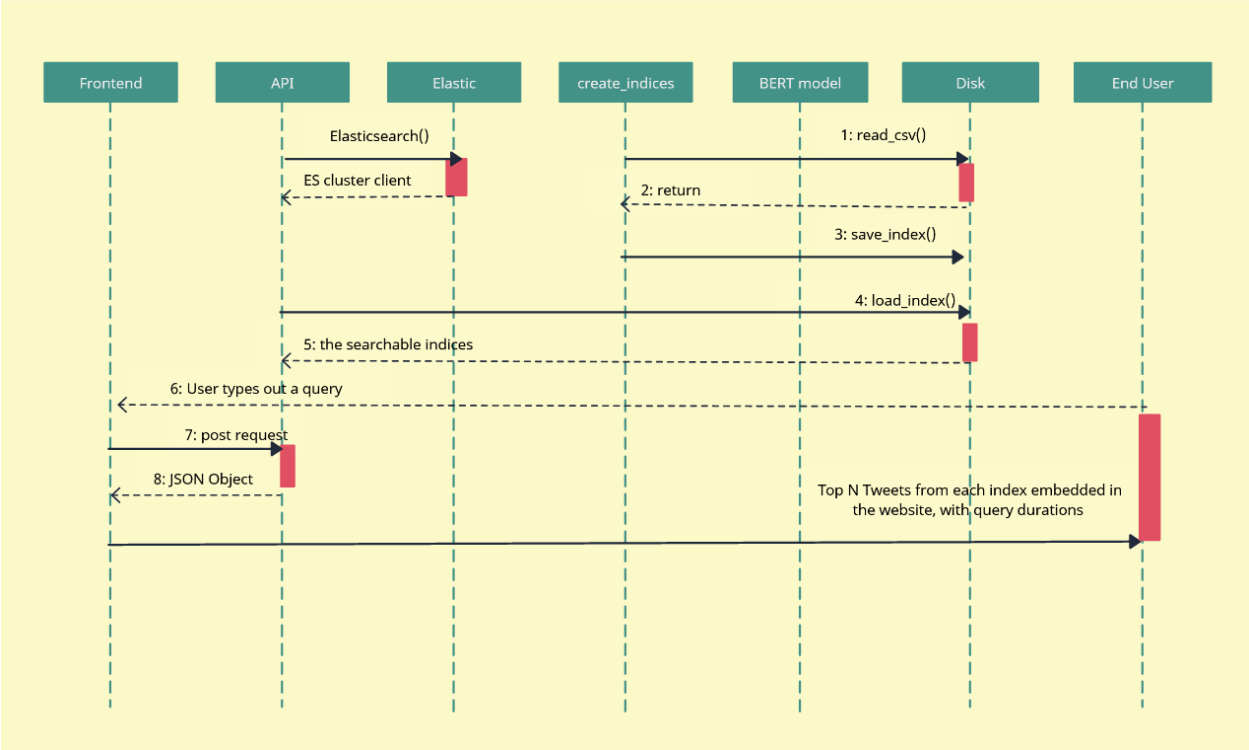

The domain model diagram that we created shows the components of our system. We first found a Twitter dataset that was a sampling of random Tweets from various users. The source we used also had datasets with images, but the size of the dataset would go from under 10gb to over 40gb when adding in images.
The ML Embedding model that we used was Google BERT's multi-qa-MiniLM-L6-cos-v1 pretrained model, which sacrifices some accuracy for embedding time efficiency. Embedded vectors are then indexed into a vector database and retrieval system. We considered using Pinecone for this part, which offers a 'vector search as a service' product, but we decided to instead create and host the indices ourselves using Facebook AI Similarity Search, or FAISS, which is used as the backend of the Pinecone system as well. FAISS offers both KNN and HNSW indices, which made it easy for us to develop both in parallel and compare the results of each. We also created a locally hosted elastic cluster using Docker with the raw tweets from the dataset, since elastic doesn't require embedding.
The retrieval systems are then all hosted and queried by our search API which was made as a Flask app and interfaces with the TypeScript and Vue frontend using post requests.

### 2.2 Product Functionality

It's important to understand the things that our product can and cannot do. It should be quite obvious, but our product does not replace the original Twitter search system. It also does not (currently) allow for images to be searched using their description although, again, with greater computing resources and time it would be possible to swap the embedding model in the embedding script and the API to a different model and feed it different data. Once embedded the rest of the system would be nearly identical, with the only change being the on-disk storage of the images (since they can't be stored in a csv) and a small impact to the KNN and HNSW retrieval time due to the increased time to embed the search query in order to search the database.
Despite these shortcomings, our product does still offer almost all of the functionality that we set out to produce. It allows for embedding of text into vector space efficiently, allowing us to deal with huge amounts of data. It allows for the embeddings and the populated indices to be stored in memory meaning the system can be spun up and down quite quickly. It allows for users to interact with the database via a well designed frontend. Most importantly, it allows for comparison between 3 different types of data retrieval both in terms of the results received as well as the time that it took to get the data.

## 3 How to use the product

On Twitter using our dataset would be a trivial process for end users since they would interact with it through the same search bar exactly as they have always done. The only difference would be on the backend and hopefully in the improved results that they see. For Twitter engineers the process would change more significantly, but the net result is still using a service to query a database to find results.
As-is, our system works a little differently. In order to run it from start to finish you would first need to find a dataset, run our embedding script, run our create_indices script, and create the elasticsearch docker container using the new dataset, all of which needs to be done only once. After those are completed, spinning up the indices is as simple as running the docker container, start the frontend using npm, and using anaconda to run the API flask app. Once there, the process is quite simple, and mostly handled on the backend.




# 4 The development process

We've already talked about the design of the system, so let's talk briefly about the development.

## 4.1 Preprocessing

Preprocessing is traditionally a large part of any NLP project, especially for Twitter data. Interestingly, BERT's sentence transformer models are trained on near raw data for the most part, which means that the rigorous preprocessing steps of stemming, lemmatization, stopword removal etc. were not necessary for our project. As a result, the preprocessing that we did was essentially just removing retweets from our dataset.

## 4.2 Embeddings

The process of creating the embeddings was quite intensive for us. Our initial research showed us that Word2Vec is the most popular 'last gen' of word embeddings, whereas sentence transformers are the more effective new wave, but have much fewer resources out explaining them because they're very cutting edge. We found Google's BERT line of pretrained models that fit our use case well - they offer everything from single to multilingual and even multimodal models, all over the spectrum of accuracy and efficiency. Again, we picked an english text only efficiency optimized model for our

project due to computational requirements for the other models and the data they would employ.
Each of these models takes a string input and outputs a 1xD vector, where D depends on the model. For our case it embeds into 384 dimensional vector space, giving us 1x384 vectors. The actual embedding process is quite simple, since our models are pretrained. We simply need to install the module, load it into our script, instantiate it with the model key that we want to use, and then call model.encode() on each document we want to embed. The difficulty with embedding is the amount of data that we need to process and store. Our dataset was 3.4 million documents, and although each document takes only a fraction of a second to encode that still comes out to dozens of hours. Luckily, we were able to employ GPU acceleration for this process which is 10x the efficiency of the embedding process.
The real issue with embeddings is storing them efficiently. You need to preserve the link between the 1x384 vector and the original text document, so the obvious idea is to put it into a pandas dataframe column. We spent a lot of time trying to figure out an effective way to accomplish this, but pandas doesn't seem to want to support it. The two best attempts that we had were to serialize and deserialize the array, but when reading the byte stream from a csv to load the embeddings it interprets it as a string, and evaluating a string to the bytes that it represents and not the bytes that it is made up of was not something we were able to figure out. The second thing that we tried was overriding the dtype of the pandas column to make it an object, then inserting the arrays individually using df.at or df.loc. This process is not possible using list comprehension or pandas functions, so we had to do it using a for loop which destroyed the efficiency of our code, making it between 5 and 20 times longer to run. What we ended up doing to solve this problem was instead of putting the embeddings into the dataframe we used the common index between the two to link the embeddings and the documents in different files. This created a new problem for us since we would run into OOM issues when dealing with the whole document at once. Our previous solution to this was that we could use pd.read_csv's chunksize keyword arg to read in only a subset of the data at a time and then write the results to a file in append mode. Nparray.save does not offer this piecewise append functionality, so we had to have two different modes for our files: the documents would be read in in chunks, embedded, added to the list of embeddings, and written to a new file while the embedding array would be held in memory for the duration of the script and only written at the end. This is possible because the whole dataset only accounts for around 4.5gb of embeddings, which is quite reasonable when you're not storing the full document dataset in memory at the same time.

### 4.3 FAISS indices

The process to populate the FAISS indices and query them was one of the easier aspects of our project surprisingly. The documentation for the library is quite bad and

ironically unsearchable for some reason, but there are enough resources out there that once you understand what to do it's only a few function calls away. FAISS supports both flatIndexL2 and flatHNSWIndex which made our job easier since we don't have to implement HNSWs ourselves. We could have optimized our indices using a variety of methods that they support, but we left that for future work.

### 4.4 Elastic

The elasticsearch component uses the elastic/elasticsearch github repo solution. This is an open source solution that works through a docker container of which you can index and get data from. The data is built using a script which sends HTTP requests to the docker container which indexes all the tweets. Afterwards, the data can be requested by pinging the /search endpoint in the backend API. This sends a request to find any content with a given keyword in the elasticsearch container.

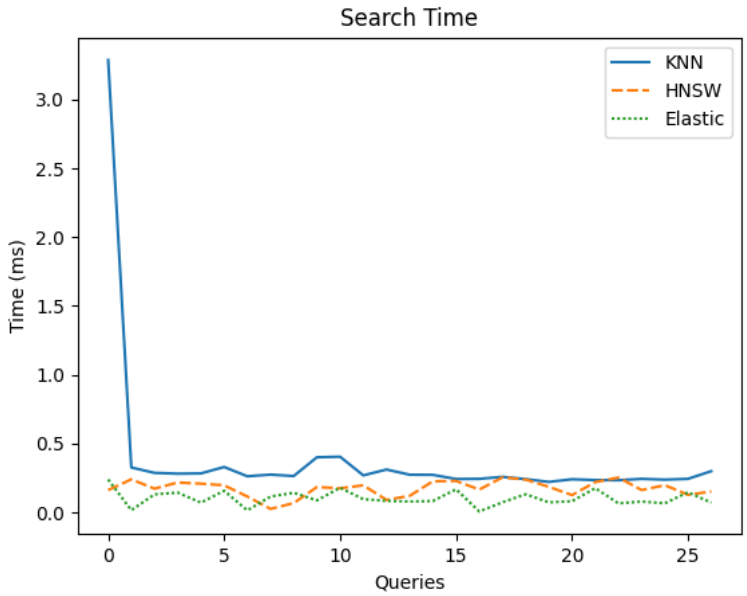
### 4.5 Frontend

The frontend was coded using the Vue.js frontend framework which consisted of Vue.js, Typescript, HTML, and CSS. Given the fact that the functionality of this project was essentially a proof of concept/test of the different search systems, we did not spend much time designing the frontend instead choosing to focus primarily on the backend. As a result, the frontend was the final thing to be completed. This was also partially because we all had limited frontend experience.

### 4.6 Testing / Results

For testing our project we decided to take 5 tweets out of our dataset, permute them in various ways, and count the successful recalls of the different systems, then create 7 random queries and qualitatively judge the results based on relevance. The results that we got are below:

| | KNN | HNSW | Elastic |
|---|---|---|---|
| Exact | 5/5 | 5/5 | 5/5 |
| Reordered | 5/5 | 4/5 | 4/5 |
| Reworded | 4/5 | 3/5 | **2/5** |
| Misspelled | **1/5** | **0/5** | 2/5 |
| Random Queries | 4/7 | 4/7 | 2/7 |




In addition to this we timed the recall of each of these and plotted them. The mean retrieval time for KNN, HNSW, and Elastic were 0.3885, 0.1752, and 0.1036 respectively. This means that elastic outperformed our other methods in speed on a smaller dataset, but HNSW did offer a dramatic improvement over KNN. HNSW also sometimes outperformed elastic, where KNN never came all too close. As far as recall, we can see that KNN performed the best out of everything for all but misspelled queries, and the two vector search offerings outperformed both random queries and reworded. All of these conclusions are easy to understand and should be expected; our FAISS indices were given the data raw and the models have no way to interpret or embed words that don't exist. Elastic, on the other hand, has built in measures like hamming score and rudimentary autocorrect that smooth the impact of having non-words in your query. On the other hand, reworded queries it had no idea what to do with since it can't draw the connections between semantically similar words. Notably, the queries that we were more aggressive with our rewording (replacing 75-100% of the words) elastic almost never got the right document back but FAISS was able to a significant amount of the time. FAISS outperforming on random queries is the only one that we didn't 100% expect, but are very happy to see.

# 5 Retrospective

There were a lot of things that went really well for us during this project. We were able to quickly learn most of the technical elements that we needed to work with throughout this project, including machine learning, api development, frontend design, web app

development, and dockerizing the elasticsearch cluster. We set our hopes quite high with the scale and complexity of the systems we wanted to make, but we were able to accomplish it all in this limited time frame. One thing that we talked about taking to other projects that we work on in the future is the softwares that we used to make sure that information was always available to every member of the team. We updated each other constantly in our group chat, we kept the code up to date in the gitlab repository, our drive contained all relevant files that any of us would need to use, and when researching and designing our system early in the semester we used a Notion page to be able to store all of the links and information that we had gathered. It takes a lot of active work to keep communication that good, but for the most part we're proud of how we kept each other up to date.

As far as things that didn't go as well, consistency is probably the easiest answer. We had several weeks where we accomplished a lot, and several weeks where little to nothing got done. If we had put in 70% effort every week instead of 100% every 2 or 3 weeks we probably would have been able to get a lot more of our reach objectives completed and been able to finish mistakes that we would have realized earlier on in the project.

We also didn't keep our dataset consistent between the two index tracks, which hurts the validity of our data. We talked about communication being a really important part, but something that we didn't really communicate was the internal logic of each individual component. Sometimes we would brief each other of the basics, but going in depth enough to understand things like the cleaning methods of the dataset wasn't something that we did. In retrospect we should have had the foresight to clean the dataset and then split the work instead of getting the dataset, splitting the work, then preprocessing it individually.

# 6 Team Contributions

- Nick - Embedding & FAISS indices
- Jonah - Front End
- Lax - Validation and Testing
- Samarth - Elastic
- Justin - Docker & Servers

# 7 Future work

Some of the things that we want to improve are

1. Syncing up the data for our indices. Currently elastic has less data due to difficulties with indexing the massive amount of data in a reasonable amount of time. FAISS has fewer issues because it can be processed locally but elastic requires data to be POSTed into it.

2. Removing retweets from the elastic index. Due to miscommunication early on that we just realized in validation we cleaned our data in different ways. As mentioned before, reindexing elastic is a time intensive process that we don't have the hours for.
3. Getting better models for FAISS, and adding images to the dataset. Again, this is really just a time consuming process that we don't have the hours for.
4. Optimizing the indices. Both elastic and FAISS have opportunities to improve response time and quality, such as stemming, lemmatizing, and stopword removal for KWS and PCA, recall training, and hyperparameter tuning for FAISS.
5. Experimenting with different preprocessing methods for both indices to see their effect. Right now we did almost nothing.
6. Making it a living database where new tweets can be added ad hoc would be really cool
7. Doing more quantitative analysis of the results of our tests.