BUILDING QA SYSTEM USING NEURAL APPROACH

Ivan Akinfiev
Department of Mathematics
akinfiev@arizona.edu

GITHUB LINK
https://github.com/ephemeraldream/NeuralWatson
https://github.com/ephemeraldream/NeuralWatson

ABSTRACT

In this article we are approaching the QA problem using neural methods. Firstly we implemented basic indexation, built basic Vector Space Model (VSM) and looked at the results. Then, the pure heuristic started. We tried different techniques to approach the optimization of search further. We used basic NLP methods such as lemmaization and stemming. Then we changed the scoring function to the LM Jelinek-Mercer Smoothing model. Finally, we implemented neural querying usnig Word2Vec technique.

0.1 Introduction

In recent years, natural language processing (NLP) has made tremendous strides, and the development of neural networks has greatly improved the capabilities of NLP models. One of the most famous examples of NLP and AI in action is IBM's Watson, which gained fame by winning a game of Jeopardy against human opponents. Building a question-answering (QA) system similar to Watson's Jeopardy search engine is a challenging task that requires sophisticated algorithms and a deep understanding of natural language processing. In this article, we will explore the process of building a neural-based QA Watson Jeopardy search engine and the techniques involved in developing an such program. We will explore how to we can extend basic VSM using neural extension. Let's start from the beginning.

0.1.1 How neural networks can help us to search better?

Neural networks can be trained using labeled data in a process known as supervised learning, where the network learns to predict outputs based on inputs with known outputs. Alternatively, unsupervised learning can be used to extract patterns and learn representations without any information about the correct output for each input. In the context of search engines, the typical workflow involves indexing and searching content, which can be done in parallel. Our task is to merge this two phenomena. Why? It is all about the efficiency and we will demonstrate it further. The integration of a search engine with a neural network is crucial as it impacts the **effectiveness** and performance of the neural search design. Even if the system is highly accurate, if it's slow, users will not want to use it. There are various methods

for integrating neural networks and search engines such as ANNs, word embeddings, attention mechanisms, transfer learning and so on. We will focus on smart word embeddings using Word2Vec model.

Typically, there is a three main directions on how one can incorporate deep learning into search engine.

- 1. **Train-then-index**: First, the neural network is trained on a collection of documents (such as texts or images). Then, the same data is indexed into the search engine, and the neural network is used alongside the search engine at search time.
- 2.]**Index-then-train**: In this method, a collection of documents is indexed into the search engine first. Then, the neural network is trained with the indexed data (and may be retrained as new data becomes available). Finally, the neural network is used in conjunction with the search engine at search time.
- 3. **Train-extract-index**: The neural network is trained on a collection of documents and then used to create useful resources that are indexed by the search engine. This approach can be particularly useful for creating embeddings or other representations that capture the meaning of documents and queries.

We will be maily focused on Train-then-Index technique called synonym query generation. We will train Word2Vec and use it as an alternative query generation. Then we will feed this query to our engine alongside the original query to improve the results. Here is a general pipleline for the engine. The picture is taken from [1].

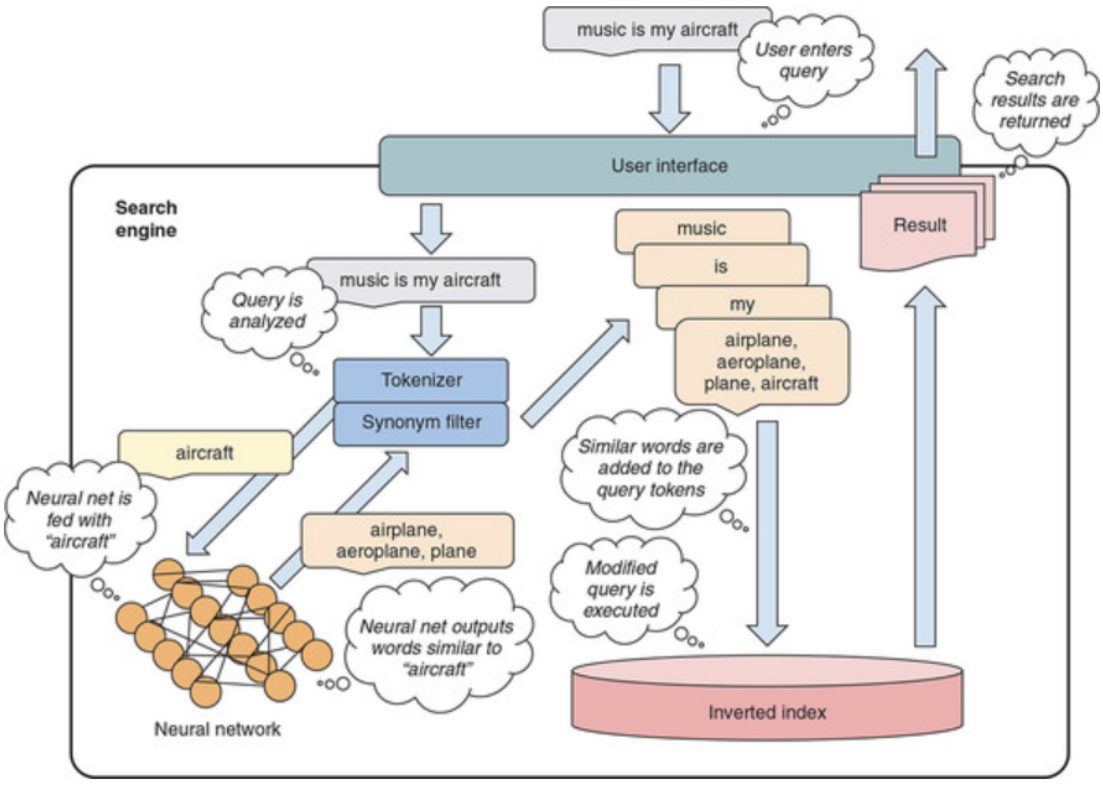

Figure 1: approxitame schema of the engine.

0.2 Preprocessing and Indexation

As with any other search engine, we must take into account the NLP side of preprocessing in indexing. Many different heuristics have been produced to build the search optimally. First, we used basic tokenization. We tried to do this with different libraries, such as StanfordNLP and OpenNLP. There was also an attempt at lemmatization and stemming from the StanfordNLP library. The choice of one or another text pre-indexing processing technique was purely empirical. We looked at MRR and decided what to do with based on the metric changes. As a result, we decided to keep only basic tokenization and stopwords elimination, since stemming almost always worsened MRR, while lemmaization interfered with the Word2Vec learning process, while practically not changing MRR too much

0.3 Scoring Function

The process of selecting the scoring function was also empirical. 4 scoring functions were proposed:

- **BM25**
- **Boolean**
- **Jelinek Mercer Model**

We chose the last one, since it outperformed the closest one with 0.10 MMR gap on average, which is an enormous overlap. We will compare performances explicitly in the results section.

0.4 Word2Vec

As we stated above, we will use neural methods to improve search results. Namely, we will create a new, synonymous queue that will run together with the original one and we will choose the best MRR of all. It remains only to understand how to generate a synonymous queue. And here, Word2Vec comes to the rescue. We will use this model in a not quite classical way:, in a sense, as an autoencoder. To begin with, we will train the model on Wikidata (when indexed), and then for each word in the queue we will generate a synonym, thereby creating something similar to the original one, but expressed through synonyms. In this way, we will construct a completely different text syntactically, which will be semantically almost identical. We have set the embedding dimension to 1000. Here is an illustration of random words in word2vec, to which I decreased the dimension to 2 using t-SNE.

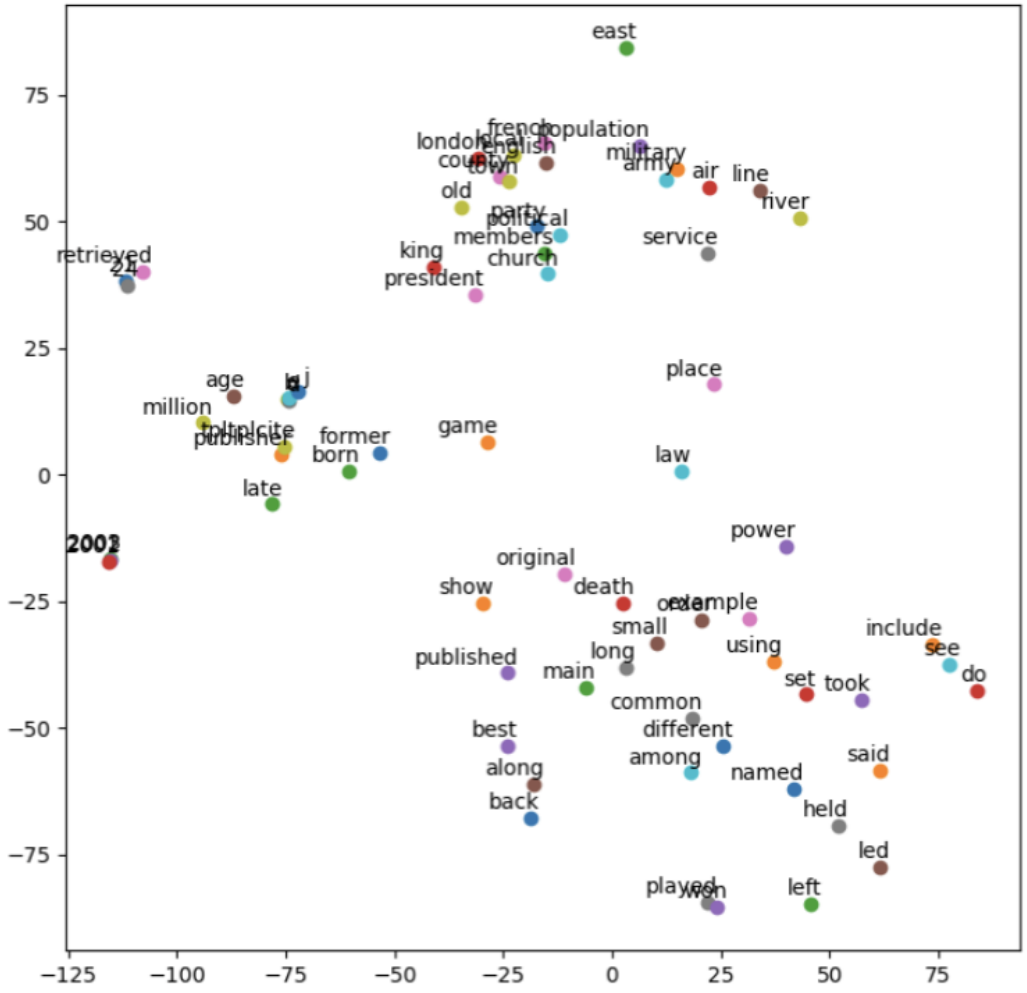

Figure 2: approxitame schema of the engine.

0.4.1 Non-Neural Approach.

We could avoid using neural methods by just swapping random words with their own synonyms and, therefore, creating more queries and run them sequentially to get best MRR. However, we have an emerging problem as we increase the amount of queries. Complexity. The total amount of all possible synonym queries is the follows:

$$\prod_{i=1}^{k}(A_i)! \tag{1}$$

where A_i is set of available synonyms for $i-th$ word in query. However,as we increase the amount of queries, the MRR also goes up. Here is a plot with polynomial regression on top of it to see the trend.

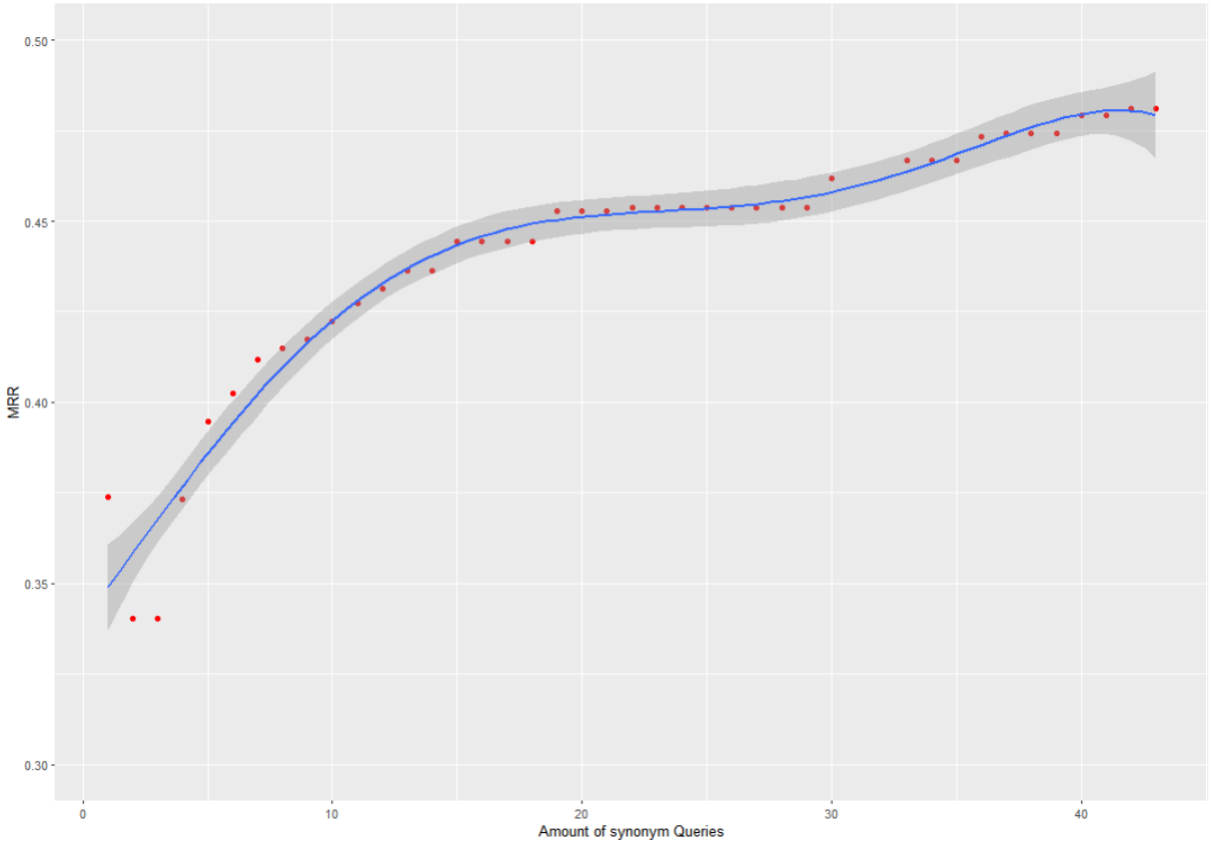

Figure 3: This plot shows the trend of MMR, as we increase the amount of queries.

As we see, the MMR flattens as we increase the amount of queries. Moreover, the runtime growth linearly along with amount of queries. So, it eventually becomes very unpractical to use. So, here the word2vec comes in handy. Only adding one neural query, we make an enormous leap by 0.2 MMR, as will be shown in the results. We implemented an engine in a way that we can combine multiple approaches.

0.5 Results

Firstly, we are going to explore how stemming, lemmaization and scoring function affects the MMR:

Processing	BM25	Boolean	Jelinek Mercer Model
Stemming	0.2142	0.171	0.26012
Lemmaization	0.2251	0.1821	0.2967
Stem + Lem	0.2014	0.19532	0.28791
No-stem + No-lem	0.21942	0.21098	0.31182

Paradoxically, the best result was achieved without lemmaization or stemming. After this analysis, we decided to remove unnecessary tags and this improved the result by 0.3 MRR. So, to work with neural synonymization, we will choose Jelinek Mercer Model with no stem/lemma. Now, we have generally two directions.

- Instead of one query, we can run more with synonym structure and hope to get more accurate result

- Directly add synonyms to the query.

We will try both approaches. Firstly, as graph shows, even 5 synonym queries + Word2Vec query brings up far better results: **0.471221 MRR** with 64 found cocuments. Adding 100 queries gives **0.48353 MRR** with 65 found documents. Using only Word2Vec without any additional queries, we have **0.381221 MRR** with 54 found documents.

Now, we can start adding synonyms to the query by simply concatenating all the synonyms. The result is the following. Using synonyms from dictionary, we have: **0.31221 MRR** with 52 found documents. Using synonyms, generated by word2vec, we have **0.3517 MRR** with 50 found documents. In conclusion, the best result achieved using brute force synonyms with multiple queries. However, We have to admit that this approach appears to be less practical for the user, since user will not be able to find the right answer. The system needs to be as much specific as it can. Adding synonyms to the query directly gets only one response which is much more transparent for the user.

0.6 Appendix: Code Documentation

Since my project involves deep learning, I need to do a lot of data pre-processing, and, therefore, I have a lot of "sleep" code. The code that was created directly to process data, which is already done, so **next classes are in sleep. There no reason to use/activate them.**

- **W2VNET** - used to create word2vec. I encountered problems with dependencies, so, to train the model, I switched to Python.
- **SynonymParser** This Class was needed to process dataset WordNET to extract the synonyms. The resulted file is stored as *parsedSynonyms.txt*
- **Serializing Questions** This files is just a parser for *questions.txt* The result is stored in two files: *SerializedQ.txt* and *SerializedA.txt* as two ArrayList<String>
- **Creating Dataset** This class parses the wikidata to create dataset for word2vec. Need to note that I didn't get a pretrained model, but trained it on the wiki-data provided.
- **Index Generator** This class was used for indexing wiki pages.

*Note:*since the eventual search system turned out to be performing better without lemma/stem. I removed it from the code.

Then we have two classes that can be used for the analysis:

- **Quering** This class was used simply for non-neural approach. Still works and can be used.
- **NeuralQuering VITAL CLASS** Everything happens here. Everything is automatized and the user can choose: scoring funciton, word2vec usage and the amount of synonym queries.

The project also contains python code. Basically, we have: 1) pre-processing of dataset, 2) open preprocessing file and training word2vec 3) matplotlib visualization of 2-D embedding with t-SNE perfomed.

- **preprocessing.py** Preprocessing of text data that we got from **CreatingDataset** class.

- **training.py** Traning word2vec and store synonym queries for QA system. Output: *W2VGeneratedSynonyms.txt*
- **visualization.py** We decrease the dimension of W2V to 2 and then visualize it.

0.6.1 Maven Dependencies

- StanfordNLP. The resuling version commented it out, so can be skipped.
- openNLP. Used for Tokenization instead of Stanford.
- Lucene latest version.

I was about to use deepleaning4j, but eventually decided not to do so and switch to pyhton for library word2vec training.

0.6.2 Python libraries used

- matplotlib
- gensim.word2vec
- numpy
- sklearn

There is no reason to run any python code except for visualization (probably). Everything is ready.

Bibliography

[1] Deep Learning for Search by Tommaso Teofili