QUESTION RETRIEVAL

Melinda Dong | a1870910@adelaide.edu.au

1. OVERVIEW

The aim of this project is to question retravel: given a question, it will be matching through the question library where store all the existing questions and return the most similar questions. It has 2 evaluation standards, to see if the ground truth duplicated question in the top 2 or top 5 most similar matchings.

3 different methods has been implemented, TF-IDF from scratch, sentence embedding based on averaging the word embedding and sentence embedding based on the thesis < A SIMPLE BUT TOUGH-TO-BEAT BASELINE FOR SEN- TENCE EMBEDDINGS >

The best result is given by sentence embedding based on the paper, it gives final top2 accuracy is 0.61 and the final top5 accuracy is 0.71.

[There are 2 submitted files:

`SearchQuestion.py` only integrate the best method (sentence embedding based on the paper) and can support the following interface

>> Python SearchQuestion.py “your question”

`question_retrieval.ipynb` described more detailed processing and experiments]

2. DATA PREPROCESSING

2.1 Data pre-processing

First read the data, the size of whole data is (363192, 6), the main columns are [question1, question2, is_duplicated], if is_duplicated is 1, means question1 and 2 are similar, vice versa. The first 100 question1 with is_duplicated == 1 are our test questions. And the whole question2 is the data we need to search from.

After drop the Nonvalue and drop the duplicated question2, 272959 question2 remained.

2.2 Text pre-processing

-- remove punctuations.

-- convert all the letters into lower case.

-- remove stop words.

-- tokenize the words using ‘word_tokenize’ package from nltk.

-- lemmatize the words using 'WordNetLemmatizer' from nltk.

[* The text pre-processing part is very similar to the previous project, more detailed reasoning please see previous project report.]

3. TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that reflects the importance of a term in a document within a corpus of documents. TF (Term Frequency) measures the frequency of a term within a document, it aims to capture the local importance of a term within a specific document. IDF (Inverse Document Frequency) measures the uniqueness of a term in a corpus. It assigns higher weights to terms that appear in fewer documents, indicating their higher importance. TF-IDF is the product of TF and IDF values. It represents the significance of the term in the document relative to the entire corpus. Terms with higher TF-IDF scores are considered more important.

Outline of the TF-IDF implementation.

- Count TF and DF

Initialize two dictionaries, TF and DF.

Iterate over the unique vocabulary list.

For each word in the vocabulary:

a. Set the initial value of DF [word] to 0.

b. Iterate over the documents (Q2) and check if the word is present in each document.

c. If the word is found in a document:

Increment DF [word] by 1 to count the number of documents containing the word.

If the word is not in TF, add it as a key and initialize its value as a list containing a tuple (doc_index, count(t,d)).

If the word is already in TF, append a new tuple (doc_index, count(t,d)) to its existing list.

- Apply logarithm on TF.

Iterate over the TF dictionary to convert the TF values into log scale. Modify each TF value to be a tuple (doc_index, log (1 + count(t,d)).

[*formula is based on lecture slides]

- Calculate the IDF

For each word in DF, set its value as N / DF [word], N is the total number of documents.

- Convert the TF and IDF values into TF-IDF

Iterate over the TF dictionary and for each word:

a. Iterate over the list of tuples (doc_index, count(t,d))

b. Multiply each TF value by the corresponding IDF value: TF (t, d) * IDF(t).

The TF dictionary now represents the inverted file, where each word is a key and its value is a list of tuples (doc_index, TF-IDF score).

Assign the TF dictionary to new name 'inverted_file' for the result.

- Using inverted file to calculate the similarity of questions.

Define the function get_top5_similar_questions that takes a query as input.

Create an empty dictionary ranking to store the similarity scores of questions.

Iterate over each word in the query:

Check if the word is present in the inverted_file.

If the word is found in the inverted_file, iterate over the list of (doc_index, TF-IDF) pairs associated with that word.

If the doc_index is not already in the ranking dictionary, add it with the corresponding score.

If the doc_index is already in the ranking dictionary, increment its score by adding the new score.

Sort the ranking dictionary items based on the scores in descending order.

Return the top 2 and top 5 items from the sorted ranking as the output.

Using the first 100 questions in question1 to test the TF-IDF model, the accuracy decreases when the search scope of question2 increase [it works good in small size of search scopes, when only matching the first 2000 questions in question2, the top 5 accuracy can above 0.9]. The whole 272959 questions as searching scope, the final top2 accuracy is 0.23 and the final top5 accuracy is 0.38.

4. SENTENCE EMBEDDING (AVERAGING)

This project is using the pre-trained word-embedding from glove. To save the computing, the version Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d) has been used. After sentence embedding, cosine similarity is used to count the degree of similarity of 2 sentence embeddings, Cosine similarity is a metric used to measure the similarity between two vectors in a high-dimensional space. It calculates the cosine of the angle between the vectors, which ranges from -1 to 1.

Word embedding is a technique used to represent words as dense vectors in a high-dimensional space, where each dimension captures a different aspect of word meaning. GloVe models consider the context in which words appear to learn word representations. Similar words are mapped to nearby vectors in the embedding space.

Outline of the sentence embedding(averaging) implementation

- Convert question1 and question2 into vectors.

Load the pre-trained glove word embedding and store them in an embedding_dict.

Create an empty dictionary Q1_dict_vec.

Iterate over each document in question1:

Initialize a zero-vector sentence vector of size 50.

Iterate over each word in document:

Check if the word is present in embeddings_dict.

If the word is found, add the corresponding embedding vector to sentence vector divided by the length of document.

Assign sentence vector to Q1_dict_vec[doc_index].

[* the way dealing with question2 is the same with question1]

- Define cosine similarity function.

cosine similarity = (A·B) / (||A|| * ||B||)

- Calculate the top5 and the accuracy.

Iterate the first 100 question1, for each test question, iterate question2 and calculate the cosine similarity scores and sort the result to get the top5 predictions from model and store in accuracy.

By averaging the word embedding to calculate the sentence embedding gives final top2 accuracy is 0.59 and the final top5 accuracy is 0.66.

5. SENTENCE EMBEDDING (BASED ON THESIS)

This part generates the sentence embedding based on the paper<A SIMPLE BUT TOUGH-TO-BEAT BASELINE FOR SEN- TENCE EMBEDDINGS >.

In the main function there are 2 innovative points. The first is using SIF (Smooth Inverse Frequency) to mitigate the impact of high-frequency words. It helps emphasize the importance of less frequent and more meaningful words in the sentence representation and good at handling out-of-vocabulary words by ignoring them during the weighting process.

The second is using SVD (Singular Value Decomposition). SVD is a matrix factorization technique that helps extract meaningful information from high-dimensional data. It subtracts the projection of sentence embedding onto the first singular vector which capture the most dominant meaning of the whole question library, which efficiently remove the global sentence-level information, allowing the sentence embedding to focus more on local and context-specific information.

Outline of the sentence embedding (based on thesis) implementation.

- Create a dictionary of unigram probability of each unique word.

Build a corpus to count how many tokens in question2 in total.

Iterate every unique word in vocabrary, for each word count how many times it appears in whole question2 in total. unigram_probability = word_count / total_words.

Store the {word: unigram_probability} as a dictionary.

- Apply SIF

The function takes four parameters: `word_embeddings` (a dictionary of word embeddings), `sentences` (a dictionary of sentences), `a` (set as 0.5 as default), and `word_probabilities` (a dictionary of word probabilities).

Initializes an empty dictionary called `sentence_embeddings` to store the embeddings of each sentence.

Iterates over each sentence in the `sentences` dictionary, within the loop, it initializes a zero vector `vs` of size 50 as the sentence embedding.

Further iterates over each word `w` in the current sentence `s`. and updates the sentence embedding `vs` by following formula:

$$vs \mathrel{+}= ((word\_embeddings[w] * a)/(a + word\_probabilities[w])) * (1/len(s))$$

And assign `vs` to the `sentence_embeddings` dictionary using the `index` as the key.

[* In case a word is not found in the `word_embeddings` or `word_probabilities` dictionary, it ignores the exception and continues]

- Apply SVD and extract first singular vector.

Creates a list `sentence_list` containing the sentence embeddings.

Initializes a matrix `X` of zeros with dimensions (embedding_dim, num_sentences).

Iterates over each embedding in `sentence_list` and assigns it to the corresponding column in the matrix `X` so that each column of X is a sentence embedding.

Performs Singular Value Decomposition (SVD) on `X` using the NumPy function `np.linalg.svd`. It returns the singular vectors `u`, `s`, `vt`.

Extracts the first singular vector `u` from the matrix `u`.

- Subtract the projection of sentence embedding onto the first singular vector.

For each sentence in `sentences`, computes the outer product of `u` with itself to create a matrix `uuT`. subtracts the product of `uuT` and `vs` from `vs`. It effectively removing the component of the embedding along the first singular vector `u`.

Assign the modified sentence embedding back to `sentence_embeddings`.

Finally, the function returns the updated `sentence_embeddings` dictionary.

- Calculate the top5 and the accuracy on test questions.

[* Still using cosine similarity to calculate the similarity, this step is same as previous sentence embedding(averaging)]

Different `a` has been tested, here is the result:

a value	0.01	0.1	0.5	1	5
Top2 accuracy	0.6	0.6	0.61	0.61	0.61
Top5 accuracy	0.71	0.7	0.71	0.7	0.7

It shows different `a` didn't affect the result too much in this case, the best result is when `a` is 0.5 and gives final top2 accuracy is 0.61 and the final top5 accuracy is 0.71.

6. CONCLUSION

Here is the comparison of the 3 methods:

method	TF-IDF	Embedding(averaging)	Embedding(thesis)
Top2 accuracy	0.23	0.59	0.61
Top5 accuracy	0.38	0.66	0.71

From the result, it shows TF-IDF has the worst performance. Using embedding

Can achieve better result and using the method introduced in paper gives the best result. Because using word embedding and cosine similarity is not affected by the length of the documents or the magnitude of the vectors. It focuses on semantic meaning, which makes it perform better in this case.

But still the overall accuracy is not high, it might because we only have 100 sample as test and the search scope is too big (272959 questions), if the search scope become smaller, it will perform better.

Here might be some of the reasons that lead the poor performance of TF-IDF in this case.

--Disregards word semantics: TF-IDF treats each word as an independent entity without considering its semantic meaning and it's unable to handle synonyms or related terms.

--Sensitivity to document length: When the document length varies significantly, the TF and IDF values may not reflect the actual importance of terms accurately.

--Vocabulary size and sparsity: As the vocabulary size increases, the dimensionality of the TF-IDF matrix grows, which can result in increased computational complexity, memory usage, and the "curse of dimensionality" problem.

--Lack of consideration for document order: TF-IDF treats documents as bags of words and doesn't consider the order of words within the document. It ignores valuable information such as word sequences, sentence structure.