Final Project: Federal Minimum Wage Increase (TREC #816)

Curtis Wilcox, Novia Wu, and Rachel Peng

Thursday, May 13, 2021

COSI 132a (Spring 2021)

Preliminary

Team member submitting code Curtis Wilcox

Title Federal Minimum Wage Increase

Description Find descriptions of the actions and reactions of either the President or Congress to increase the U.S. federal minimum wage.

Narrative Relevant documents include descriptions of advocacy or actions (or lack thereof) taken by the President or Congress to increase the U.S. federal minimum wage, including increases for government contract workers. Analyses and discussions of pros and cons of an increase by talking heads is not relevant.

Queries

- **Updated Description:** actions and reactions of President or Congress to increase U.S. federal minimum wage
- **Updated Narrative:** advocacy or actions (or lack thereof) by the President or Congress to increase the U.S. federal minimum wage, government contract workers

(Brief) Summary Our team did multiple things to optimize the user's retrieval of documents that are pertinent to this subject.

We created two new custom analyzers – one that used the `trigram` tokenizer and one that used the `whitespace` tokenizer (both analyzers have filters based on lowercase letters, stopwords, asciifolding, and use the porter stemmer).

We implemented a synonym mechanism to invoke query expansion (if the query has fewer than four words). We also implemented query summarization using `spaCy` (if the query has greater than seven words).

We removed boilerplate text from queries and utilized the set difference between the results of documents from a relevant query and the results from an irrelevant query (for example, anything about talking heads) to remove irrelevant documents from the overall result set.

We attempted to use the `TF-IDF` scoring mechanism to reduce the number of irrelevant tokens in the embeddings in hopes of improving the retrieval performance (especially for `fastText`). Unfortunately, this proved too complicated for us.

The Flask web app will automatically suggest queries for the user based on what they have typed in at any given moment. The suggestions are based off of the titles of the documents.

Files to Look At

- `fp.py`
- `evaluate.py`
- `es_service/doc_template.py` and `es_service/index.py` (only to change building the index – adding new fields)
- `tfidf.py`
- the `HTML` files (if you feel so inclined)

Note All of our files are based off of `fp_data/` (similar to how the homework assignments were `paX_data/`, where `X` is the assignment number).

Description

Flask Application This program allows a user to enter a search query and filter through several thousand documents by seeing which documents contains any of the terms searched for (with some exceptions). The user can see eight results per page (where each result is the title and a snippet of the document's content) and navigate through the pages, and also click on links to each document's full content.

Command Line Interface (CLI) There is also a provided CLI. The user can enter queries to obtain the `Normalized Discount Cumulative Gain` (`NDCG`) score and the `Average Precision` of a query by specifying the following information:

- `--index_name`: the name of the index;
- `--topic_id`: the ID number of the topic;
- `--query_type`: the query (which will be the `title`, `description`, or `narration` of the topic specified);
- `--vector_name`: optionally, the vector to use for reranking (`sbert_vector` or `ft_vector`);
- `--top_k`: the number of documents to retrieve; and
- `--analyzer`: which analyzer to use (either `default`, `n_gram`, or `whitespace`).

Dependencies

- `Python 3` (specifically, `Python 3.8`)
  - Installed when creating the virtual environment (venv). Using `miniconda`, the command was `conda create -n cosi132a python=3.8`
- `elasticsearch, elasticsearch-dsl, sentence-transformers, flask, numpy, zmq, unidecode`
  - Installed using `pip` in the activated venv (`pip install -r requirements.txt`, where `requirements.txt` contains each of the previously mentioned requirements, separated by a newline, and nothing else)
- `spaCy`
  - On the command line, run the following commands (within the environment):
  - ```
    pip install -U pip setuptools wheel
    pip install -U spacy
    python -m spacy download en_core_web_sm
    ```

Build Instructions

0. Note: Steps 1-8 should be run on the command line. Please also note that these steps are tried-and-true on MacOS. Nobody on this team uses Windows or Linux, so we apologize if they do not work on those machines.
1. `conda create -n my_environment python=3.8  # my_environment could be anything you want`
2. `conda activate my_environment`
3. `pip install -r requirements.txt  # requirements.txt should be provided`
4. `python load_es_index.py --index_name wapo_docs_50k --wapo_path fp_data/subset_wapo_50k_sbert_ft_filtered.jl`
5. `python -m embedding_service.server --embedding sbert  --model msmarco-distilbert-base-v3`
6. `python -m embedding_service.server --embedding fasttext --model fp_data/wiki-news-300d-1M-subword.vec`
7. `python fp.py --run`
8. Navigate to `localhost:5000` in your browser of choice.
    ```

Run Instructions

Flask Application

1. Enter any string into the search bar provided at the top of the screen and select which search method you would like to utilize, followed by the number of results you want returned. Press the `Search` button or hit the `return` button on your keyboard when you are satisfied with your query.
2. A maximum of eight results will be displayed on a page. If your search query had more than eight hits, there will be a `Next Page` button at the bottom that can be pressed to navigate to the next page.
    1. If you enter nothing into your query, you will receive no results.
    2. There is no button on the website to go backwards: you must use the browser's `back` button to move in that direction.
    3. The documents will be displayed in order of supposed relevancy.
3. Each result shows the title of the document that matched and the first 150 characters of the document. To see more information or content of a specific document, click on the title (as it is a link) to take you to the appropriate page.
4. If you wish to enter a new query, you need only enter a new query in the search bar at the top of the results page. There is no search bar when viewing information specific to one document. For convenience, the "active" query will be presented in the search bar, but you need only change the text and run a new query for new results.

CLI

1. Start the query with `python evaluate.py` (the venv from above should of course be activated).
2. After specifying the filename, add the arguments that are explained in the `Description / CLI` section above.
3. View the data at your leisure!

System Design Description The system that we created for this final project runs primarily off of two files. The `fp.py` file runs the Flask app, and it contains logic for obtaining information that the system will utilize (such as documents to filter through, select, and read) and passing it to the relevant recipients (that is, the HTML pages seen by the user). When the documents are retrieved, they are held in a dictionary that maps a list of documents to the page number that they will appear on. The `evaluate.py` file runs the CLI.

`utils.py` contains the logic for reading in the information in the documents (and extracting the information requested by the assignment). That would be reading in the `wapo` data from the `.db` file and also the wapo topics from the `.xml` file. There is also a helper function for taking in an iterable and returning the first unique `n` elements that have a length greater than `min_length`.

`evaluate.py` contains the logic for a command-line interface that will display the `NDCG@{top_k}` score for a query. The data can be queried by using the title, description, or narration of a specified wapo topic. The script must also be supplied the number of documents to return and whether the default analyzer or the custom analyzer will be used, as well as whether or not the results should be reranked (using the `sBERT` embeddings or the `fastText` embeddings if so). More information is provided in `Description/CLI` and `Run Instructions/CLI`.

`load_es_index.py` reads in the wapo data and constructs the index for accessing its information.

`metrics.py` contains logic for calculuating the `NDCG` score and the `Average Precision` of a query.

`tfidf.py` gets the information for calculating the `TF-IDF` scores by getting raw term and document frequencies, and then pickles them. As noted elsewhere, the work in this file unfortunately did not come to fruition.

`home.html` is the "index" file for the site. It presents a search bar to the user and allows them to enter a query to find relevant documents, by way of selecting the search method (reranking, which analyzer to use, number of results to retrieve).

`results.html` displays a list of documents that match the query. A maximum of eight results can be displayed at any one time, so if there are more than eight results there will be a "next page" button for the user to click to navigate through the list. The same search bar is displayed to find other documents. There is no "previous page" button for the user (instead, they may use their browser's "back" button). Each result is comprised of two parts:

the first is the title of the document in a link format, so that the user may click on it to access the document's full information; the second is the first 150 characters of the document's content, followed by an ellipse (...). Each result is numbered `1` through however many results are returned.

`doc.html` displays specific information for a selected document (the title, the author, the date the document was published, and the content). The only way to navigate back is by using the browser's back button.

Output

Baseline Results (with only the default analyzer)

NDCG	Title	Description	Narration
BM25 / Default Analyzer	0.767	0.836	0.708
sBERT / Default Analyzer	0.783	0.832	0.858
fastText / Default Analyzer	0.579	0.628	0.676

Average Precision	Title	Description	Narration
BM25 / Default Analyzer	0.476	0.803	0.570
sBERT / Default Analyzer	0.709	0.717	0.832
fastText / Default Analyzer	0.331	0.502	0.457

After Results (with the default analyzer and the two new custom analyzers)

NDCG	Title	Description	Narration
BM25 / Default Analyzer	0.836	0.879	0.757
BM25 / n_gram Analyzer	0.385	0.731	0.634
BM25 / Whitespace Analyzer	0.597	0.768	0.680
sBERT / Default Analyzer	0.857	0.896	0.822
sBERT / n_gram Analyzer	0.859	0.844	0.700
sBERT / Whitespace Analyzer	0.842	0.839	0.716
fastText / Default Analyzer	0.605	0.783	0.817
fastText / n_gram Analyzer	0.332	0.870	0.895
fastText / Whitespace Analyzer	0.430	0.958	0.865

Average Precision	Title	Description	Narration
BM25 / Default Analyzer	0.527	0.921	0.613
BM25 / n_gram Analyzer	0.238	0.638	0.526
BM25 / Whitespace Analyzer	0.437	0.731	0.649
sBERT / Default Analyzer	0.787	0.944	0.715
sBERT / n_gram Analyzer	1.0	0.864	0.656
sBERT / Whitespace Analyzer	0.877	0.832	0.742
fastText / Default Analyzer	0.344	0.724	0.698
fastText / n_gram Analyzer	0.153	0.640	0.686
fastText / Whitespace Analyzer	0.266	0.840	0.724

Breakdown by Teammate (Including Teammate-Specific Results)

Curtis I predominantly worked on the autosuggestion feature demonstrated in the Flask application. This was more difficult than I had originally anticipated because of the shortage of clear documentation. There were a healthy amount of examples on the `ElasticSearch` site, but they were all `cURL` examples and not `ElasticSearch DSL` (`DSL`) examples, and I did not find it terribly easy to parse from `cURL` to `DSL` (that is to say, nigh-impossible). I eventually determined that a new field needed to be added to the `BaseDoc` class (of type `elasticsearch_dsl.Completion`), and that a field needed to be added when constructing the index.

The field added to the index needed to be a list of strings (where each string was a suggestion for the document), which I decided to create by removing non-alphabetic (or space) characters from the title, then taking the first six words that are unique and not one character long, and generating all permutations of the words in that string. Unfortunately, the index now takes about 20 minutes to build.

In the HTML files that include a search bar (`home.html` and `results.html`), there is an additional `JavaScript / AJAX` script included in the `head`, as well as two `jquery` scripts. The script adds an autocomplete aspect to the search bar where the user enters, and handles sending the data to the `search` function, as well as getting the data back from the search function and sending it to the provided `autocomplete` functionality. This website-specific logic was taken from a website cited in the `.html` files and slightly adapted to fit the needs of this assignment.

The Flask-side display of success comes in the form of the `search` function in `fp.py`. The search term is obtained from the `HTML` form and a connection is made to ElasticSearch. Then use the `Search.suggest` method, pass it a title to return the suggestions under, the term to search over, and a `completion` argument that specifies that the field with information for completing the autocompletion in the form of a dictionary (e.g., `{'field': 'field_from_doc_template'}`). The responses are then "jsonified" (`jsonify` is a Flask method that wraps over Python's `json.dumps` to turn the result into a `flask.Response`) and sent back to the web-server side of things.

Ideally, the suggestions would not be based solely on the first six unique two-or-more-character words in the title of the document, and have more to do with parts of the content of the document being suggested. However, since generating permutations is expensive, and it took so long to paw through layers of difficult documentation, StackOverflow questions, and GitHub Issues, and get all of the parts to work together, this is the method that was selected.

Time Spent I spent about fifteen hours into this assignment.

Difficulty The difficult part of this assignment was twofold. First, the expectations were not clear and we didn't have very much to go off of, which made starting and gaining any sort of momentum quite tricky. Second, the documentation of ElasticSearch, while seemingly extensive, had examples that were useless to us because we are using the `elasticsearch_dsl` Python wrapper, which in my opinion does not have useful documentation.

General Comments I think this final project assignment could use some serious refinement. I understand that it hasn't been given before, and I don't think that giving it during a remote (pandemic) semester was the best call.

Novia

Tasks

- Removal of boilerplate texts in queries.
- Perform set difference of relevant query search and irrelevant query search to eliminate irrelevant documents.
- Ran most of the scores, before improvement and after each feature improvement.
- Computing `TF-IDF` scores for each token to try to filter out less important words, so only important words are embedded.
- Logic and code implementation:
  - Simple removal of the boilerplate texts in the given TREC queries. I thought about doing automation on that but Jingxuan mentioned it's just for this specific topic so I implemented a list of custom queries in the `evaluate.py`. The performance of ElasticSearch improved a lot once you removed them, implying that query formulation and processing is essential to retrieval.
  - In the `evaluate.py`, I implemented a set difference algorithm in hope to reduce the number of irrelevant documents returned. The basic logic is that we use our original query to search like usual with all improvements, then we search for the irrelevant query. Then for the documents that got retrieved in the relevant document list, if it also showed up on the irrelevant list, then it might be irrelevant and I do not display those irrelevant documents, they are also not counted in the calculation of the `NDCG` score.

- Running scores on different stages of improvement that we implemented, nothing fancy here.
- Attempted to compute the `TF-IDF` score to improve the fastText embedding. I wanted to compute all the term frequency and document frequency, and then be able to compute the `TF-IDF` scores for each token. Then based on the average of the `TF-IDF` score, I would come up with a threshold, and any words below the threshold `TF-IDF` score will not have embedding. This way hopefully the fastText embedding will be more informative. The code is changed on the `text_processing.py` and embed.py, also added a file called `TF-IDF` that goes through the corpus to get the necessary information for the calculation of the `TF-IDF`. The code change in the files except for `tfidf.py` is commented out because of system errors. Unfortunately I encountered many difficulties trying to implement the whole advancement of the embedding service.

Time Spent I spent around 13 hours researching, asking questions, implementing, scoring the retrieval methods.

Difficulty

- Researching what is out there to improve the retrieval method took a lot of time, we were very confused in the beginning because of lack of experience in the field.
- Trying to figure out `TF-IDF` for the embedding is very difficult. I was trying to improve by getting the `TF-IDF` score for all the tokens, and then set a threshold, and "eliminate" the words that have little significance to the documents, in hope to increase the accuracy by leaving more important words. However it did not work out. I understood theoretically how to compute and `TF-IDF`, however, since the implementation is mainly in the `embedding_service` that is very complicated, even after asking Jingxuan for help multiple times, I could not fix all the bugs that the system is telling me. I spent a lot of time working on this part but unfortunately it did not work out.

Rachel Besides doing researches to gain future directions, I focused on working on two improvements for our model.

- Created two custom ElasticSearch analyzers:
  - N-gram: this analyzer has `min_gram` of 3 and `max_gram` of 4. The goal for using this is to test if the `n-gram` language model improves metrics and ranking.
  - Whitespace: this analyzer breaks text into terms whenever it encounters a whitespace character. The goal for using this is to have a basic analyzer that serves as a baseline. This can allow us to compare n-gram with the baseline.
- Implemented query optimizations: while it is important to work on improving the embedding models, having refined user queries can also improve retrieval performance. In simple words, we think that there are two scenarios that a query that needs to be optimized: when it is either too short or too long. We first remove all the punctuations, all the stopwords, and normalize the tokens for all queries. Then, depending on the length of the query, we decide to do query expansion if there are more than eight tokens; if there are less than three words, we do text summarization.
  - Query expansion: this is done by finding synonyms for each token in the query using NLTK.WordNet. WordNet nicely returns all the synonyms of a word, and I remove all with an underscore, as they will indefinitely add noise to the model. The synonyms are then appended to the original query to form a longer query. We acknowledge that this may add noise to the model, but we did see that query expansion reranks more relevant documents more to the top.
  - Text summary: This is done through `spaCy`, a python package for advanced natural language processing. I used `spaCy` to get the name entity recognition. For example, "SpongeBob SquarePants" will be recognized as a PERSON, while "FBI" will be an ORG. This would be helpful to identify keywords in long queries, since information such as the name of a person or organization, geographic locations, and numbers are crucially important, as they will reduce the noise level, which is high for long queries. I defined keywords to be tokens that have either 'PERSON', 'GPE', 'NORP', 'ORG', 'TIME', 'CARDINAL', 'MONEY', 'EVENT' as their name entity recognition. In addition, I manually initiated two lists to ensure important words are not excluded from the process. The first contains all the keywords from topic 816's title, description, and narrative (mostly nouns and distinguishable verbs). The second is a list of state names, with abbreviations, as geographic locations are distinguishable information, and the minimum wage is

different from state to state. As a result, the query expansion reranks more relevant documents more to the top and brought up significantly more relevant documents.

- Also, I implemented a naive system to filter out the less significant words by setting a term frequency threshold. First, the frequencies for all tokens in the query are calculated and stored in a dictionary. Then, the threshold is calculated by multiplying the average frequency by a constant scalar. We used `1.2` in the code, but the higher the threshold, the fewer the words are left. However, this is a naive approach because one, the most frequent word in a query may not be informative (i.e. person), second, query token frequency needs to be calculated using the token document (i.e. include query token that appears more frequently in the documents).

Time I spent around 13-15 hrs on this project. Researching and debugging took a lot of time.

Difficulty I learned a lot from this project, but I think it's challenging in the way that I didn't know what to research from the beginning, and I was stuck to find a place to begin.