Course Project: Automated Fact Checking

Kristen Hayzelden
Individual Report

1 Introduction

The abundance of misinformation online has sparked an interest in automated fact checking software/methods. This coursework explores information retrieval and data mining methods to assess the accuracy of various claims using the Fact Extraction and Verification (FEVER) dataset.

General note: The coursework specifies to get the details / top documents for the first ten claims in the "train.jsonl" file. However, the first ten claims do not align with the listed claim ID's of "75397, 150448, 214861, 156709, 129629, 33078, 6744, 226034, 40190, 76253". The first ten claim ID's are actually "75397, 150448, 214861, 156709, 83235, 129629, 149579, 229289, 33078, 6744". This means that in lieu of claims 226034, 40190, and 76253, I have used claims 83235, 149579, and 229289.

2 Task 1: Text Statistics

2.1 Term Frequency

To count the term frequency for this document set, I used the following code to load every jsonl file and for each file load each line as its own json. This resulted in loading over 5 million documents as the full dataset.

```
json_files = [j for j in os.listdir(path_to_jsons) if j.
    endswith('.jsonl')]

for path in json_files:
    newPath = 'data/wiki-pages/wiki-pages/' + path
    with open(newPath, 'r') as f:
        for line in f:
            data.append(json.loads(line))
```

Once this data had been loaded into my program, I took the body of each document (defined as "text") and removed all non-alphanumeric characters for easier processing. Once the offending characters had been removed, I concatenated the bodies of all the documents into one varabile. This allowed me to process all text bodies at once which was the most efficient method for the purpose of this task.

```
from nltk.tokenize import word_tokenize

words = word_tokenize(text)
```

Using the variable I had created, which contained the filtered text of the entire corpus (appropriately named "text"), I then used the natural language tool kit python library to tokenize the text into an array of words. The NLTK is a python library designed to aid developers in working with natural language data. By using NLTK, I was able to receive a list of all tokens in the text – this does not filter any words for this task. The removal of stop words (words such as "the", "a", "and", etc. that do not provide contextual purpose and are essentially filler words) would not be appropriate for this task and is discussed further in task 2.

The next step is to count the words. The NLTK tokenization process creates a list of every word in the data. As lists do not have limitations regarding duplications, it contains each word in the order it appears in the text. Therefore, I needed to convert the list into a dictionary. This process removes duplications as dictionaries in Python do not allow duplicate keys. With the dictionary of unique tokens and the list of all tokens, I was then able to iterate through the initial list and adjust the dictionary's value as each word appeared. This resulted in a dictionary with key-value pairs of each token and each token's number of occurrences.

```
wordsTF = dict.fromkeys(words, 0)

for w in words:
    wordsTF[w] += 1
```

This resulted in data showing what we already know to be true. The top ten most frequent words are all what we would consider "stop words". These filler words are in every document and for our purposes do not provide any useful information. It is also worth noting that "rrb" and "lrb" are used in the data set to denote "(" and ")" respectively.

Word	Number of Occurances
the	30,473,045
of	15,936,679
in	14,395,680
and	12,584,719
a	10,705,457
is	7,813,064
to	6,616,507
was	5,951,941
rrb	5,937,490
lrb	5,936,530

2.2 Zipf's Law

Using the information from the term frequency calculations, I was then able to plot the data to demonstrate Zipf's law for this dataset. Zipf's law states that in a corpus of text, a word's frequency is inversely proportional to its frequency rank. This means that the highest ranked word will appear twice as the word below it in order of frequency and three times as much as the next and four times as much as the next and so on.

To plot this dataset, I used a common scaling procedure for Zipf's law, where the x-axis of the figure is the log of the rank of the term in regards to frequency (i.e. "the" is ranked at number one as it has the most occurrences) and the y-axis of the figure is the log of the frequency for that word. The log is used to help scale the data. To plot this data, I used python's Pyplot.

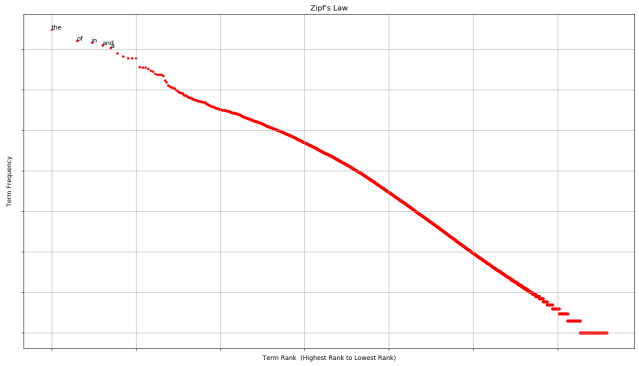

Figure 1: The plot for Zipf's law for this dataset.

3 Task 2: Vector Space Document Retrieval

3.1 TF-IDF of the 10 Claims

As the coursework specified, the only words for which I calculated the TF-IDF were the words that have impact on the claims.

Once I had loaded the documents and the first ten claims, I looped through each claim to calculate the information required. Within this loop I performed many sub tasks. For each claim I removed all the non-alphanumeric characters, tokenized the claim, then removed the stop words from the token list. The removal of stop words is pertinent to this task as it removed those words discussed earlier ("the", "a", "and", etc.) that will not help in determining relevant documents for the claims. Once this information has been filtered, I then count the words occurrences within the claim itself. This is used later as part of my normalization calculations. This helps balance the claims regarding term importance. If a claim were to have a (non stop) word multiple times, then that word is clearly important.

For normalization, I used the log method. Following this method, I normalize the term frequency of each word of the claim with the following formula:

$$TF(w) = 1 + log_{10}(count(w, c))$$

Figure 2: TF calculation where "w" is each word in a claim and "c" is the claim.

Still within the loop of each claim, I then loop through each document and count the number of documents that contain each word. To ensure the counts are correct, I do filter the body of each document for non-alphanumeric characters and all text is compared as all lowercase. A separate dictionary stores each word and how many documents contained that word.

```
for word in wordsIDF:
    for entry in data:

        temp_text = re.sub(r'[^a-zA-Z\d\s]', "", entry['
    text'])

        if word in str(temp_text).lower():
            wordsIDF[word] += 1
```

Following this count, I calculate the IDF using the following formula and code:

$$log_{10}(\frac{\text{total number of documents}}{\text{number of documents containing } w})$$

Figure 3: The formula for IDF, where "w" is the given term.

```
for word in wordsIDF:
    if wordsIDF[word] != 0:
        wordsIDF[word] = math.log10(float(len(data)) / (
    float(wordsIDF[word])))
```

Note that if the number of documents that contain the word is zero then the calculation is skipped. The value remains at zero and prevents zero division. This completes the IDF calculation for each claim. Now that I have this information and the TF for each word, I then can calculate the TF-IDF for each term in the given claim. The process for this is simple enough, where I take the product of each word's TF and IDF. However, the process is not complete as the TF-IDF needs to be normalized to ensure the values are proportionately represented.

```
for word, tfidf in wordsTFIDF.items():
    normalization_denom += tfidf ** 2

normalization_denom = math.sqrt(normalization_denom)

for word, tfidf in wordsTFIDF.items():
    wordsTFIDF[word] = tfidf / normalization_denom
```

The cosine normalization process is shown above. To do so, I took the TF-IDF of each word in the claim and square it. Then with the sum of these TF-IDF's, I get the square root. This value is used as a denominator in a normalization process. For each word in the claim, this value is used to normalize the TF-IDF of said word. This is following the formula depicted below. For each word in the claim, you divide the TF-IDF by the denominator explained above.

$$\forall w : \frac{w_{1\ldots n}}{\sqrt{{w_1}^2 + {w_2}^2 + {w_3}^2 \ldots {w_n}^2}}$$

Figure 4: The cosine similarity formula, which is repeated for each word in a claim.

This entire process repeats for each claim and results in the normalized TF-IDF of each word in each claim. The end result is a dictionary with the ID of the claim as the key and a second dictionary as the value. The second dictionary is a list of all tokens and the claim's TF-IDF value for that token.

3.2 TF-IDF of Documents

Following the same process as described for the claims, I am able to calculate the TF-IDF for each document. As it was specified in the coursework, we are only required to determine this for words that would affect the claims. Therefore, I kept a list of each token in the claims – allowing me to only calculate the TF-IDF for the relevant words.

As the TF for each document varies, I loop through all the documents and count the occurrences of the claim tokens. Using this TF, I then calculate the TF-IDF with the previously calculated IDF values. The IDF values remain the same for the entire corpus so they are reused from the initial calculation to save computation power. The TF-IDF is then normalized in the same way as the claim's TF-IDF.

The end result is a dictionary with the ID of the document as the key and a second dictionary as the value. The second dictionary is a list of all tokens and the document's TF-IDF value for that token.

3.3 Cosine Similarity

```
for documentID in documentTFIDF.keys():
    for claimID in claimsIDF.keys():
        simularity_score = 0

        for word, word_tfidf in claimsTFIDF[claimID][0].
    items():
            simularity_score += documentTFIDF[documentID
    ][0][word] * word_tfidf
```

With the normalized TF-IDF's, the cosine similarity calculations were simple. For each document I loop through each claim and within that claim check each word. The similarity score between a given claim and document is the sum of the products of each word's TF-IDF in the document and in the claim. This task is simple due to the normalization steps taken in the TF-IDF calculation process.

$$SIM[c_n, d_n] = \sum_{w \in c_n} c_n \text{ TFIDF } w * d_n \text{ TFIDF } w$$

Figure 5: The cosine similarity formula, where "c" is any claim "d" is any document.

3.4 Top 5 Documents

Following the cosine similarity calculations, I then take the top five scoring documents for each claim. This should, in theory, result in the top five documents that are capable of proving the claim true. My results do seem to hold true based on a manual evaluation. For example, claim "33078" is "The Boston Celtics play their home games at TD Garden."

When looking at this claim's top five documents, the titles alone seem promising. But upon a manual evaluation of these Wikipedia entries, the evidence is clear that this claim is true and these documents can support this claim.

Rank	Document ID	Similarity Score
1	TD_Garden	0.937148499
2	Boston_Garden	0.936225752
3	1974 - 75_Boston_Celtics_season	0.934109068
4	Boston_Celtics	0.929491329
5	1985 - 86_Boston_Celtics_season	0.922367557

4 Task 3: Probabilistic Document Retrieval

4.1 Query-Likelihood Unigram Language Model

$$\sum_{w \in q} count(w, q) * log_{10}(\frac{count(w, d)}{|d|})$$

Figure 6: The query-liklihood unigram language model employed. Where "w" is each word in a claim, "q" is the claim, and "d" is the document.

The query-likelihood model I built was based off most of my code from previous tasks. For the model, I needed the probability of the claim's words to occur in a given document. To determine the probability, one simply can take the number of occurrences of the word and divide by the total number of words in that document. As this is a unigram language model, I used the code from the previous task to carry out this function. However, slight modifications were required. Namely, if a word had zero occurances in a document – the count was altered to be "0.00001" in lieu of a solid 0 count. This is to balance the likelihood.

As the query-likelihood formula I use relies on the product of each word's likelihood – if any likelihood was left as zero it would result in the entire document receiving a zero score for that claim. This means that without this adjustment, a document that had nine of ten claim words could receive a score of zero due to the one missing word.

Once the adjustment is made and the probability calculated the query-likelihood with the above formula. The code is as follows:

```
for word in words_count:
    word_probability = 0
        occurances = temp_text.lower().count(word)

        if occurances != 0 :
            word_probability = float(occurances)/float(
    length)
        else:
            word_probability = 0.00001 # hardcoded value
    to replace 0

    total_probability += (words_count[word] * math.log10(
     word_probability))
```

This value is saved for each claim-document duo and then the top five are extracted from the list in the same was as done in task 2. From here we can see the results are still promising. For example, look at the same claim as discussed previously, "33078: The Boston Celtics play their home games at TD Garden.". The results are good, in that when manually investigated they do prove the claim to be true but some strange results seem to be present when compared to the more consistent results from task 2.

4.2 Smoothing

4.2.1 Laplace Smoothing The first smoothing method implemented was Laplace smoothing, also known as additive smoothing. This method simply takes the previous methods but adds one to the occurrence count of a word. I also add the length of the vocabulary

Rank	Document ID	Similarity Score
1	1974–75_Boston_Celtics_season	-14.050823
2	2006_Hockey_East_Men's ...	-15.050688
3	2007_Hockey_East_Men's ...	-15.050688
4	Boston_Celtics	-15.322656
5	KDrew	-15.837157

of all claims to the denominator. Therefore, the claim's similarity score with a given document will be the result of the following formula.

$$\sum_{w \in q} \frac{count(w, d) + 1}{|d| + |v|}$$

Figure 7: The Laplace smoothing formula, where "w" is a word in a claim, "c" is the claim, "d" is a document, and "v" is the vocabulary set of all claims.

Based off of my manual investigation, this method did not produce as high quality results as the previous method. For example, look at the same claim as discussed previously, "33078: The Boston Celtics play their home games at TD Garden."

Rank	Document ID	Similarity Score
1	Fruitdale	0.40540540
2	Tabletop_game	0.40540540
3	Homegrown_Player_Rule	0.40540540
4	South_Boston	0.39999999
5	George_Home	0.38709677

It seems that only two of the results provide support to prove this claim with this method. This, however, could be due to developer error.

4.2.2 Jenlinek-Mercer Smoothing The second smoothing method implemented was Jenlinek-Mercer smoothing. This method employs a value, lambda, which is used to smooth the results. This value is not set specifically and is to be tweaked for each dataset. For this project I have set lambda at a value of "0.5".

$$\sum_{w \in q} \lambda(\frac{count(w, d)}{|d|}) + (1 - \lambda)(\frac{count(w, a)}{|a|})$$

Figure 8: The Jenlinek-Mercer smoothing formula, where "w" is a word in a claim, "c" is the claim, "d" is a document, and "a" is the entire corpus.

The code to achieve this result is as follows:

```
for word in words_count:

    word_probability = 0
    occurances_document = temp_text.lower().count(word)
    occurences_corpus = counts[word]

    if document_length != 0 :
        word_probability = (0.5 * (occurances_document/
    document_length)) + ((1-0.5) * (occurences_corpus/
    corpus_length))

    total_probability += word_probability
```

Again, looking at the same claim as discussed previously, "33078: The Boston Celtics play their home games at TD Garden." The results are better than the additive smoothing, but not as accurate as my initial results.

Rank	Document ID	Similarity Score
1	List_of_role-playing...	0.33541937748326683
2	Forplay	0.2878003298642192
3	LTD	0.21637175843564777
4	TD	0.21637175843564777
5	Teegarden	0.1895860441499335

4.2.3 Dirichlet Smoothing The final smoothing measure employed was Dirichlet smoothing, where the method is the same as Jenlinek-Mercer smoothing, but the value of lambda is calculated instead of being arbitrarily set. Lambda, in this case, is set to the average document length. This is an attempt to average out counts so documents of varying lengths are still treated properly.

$$\sum_{w \in q} (\frac{|d|}{|d| + u}) * (\frac{count(w, d)}{|d|}) + (\frac{u}{|d| + u}) * (\frac{count(w, a)}{|a|})$$

Figure 9: The Dirichlet smoothing formula, where "w" is a word in a claim, "c" is the claim, "d" is a document, "u" is the average length of all documents in the corpus, and "a" is the entire corpus.

Finally, looking at the same claim as discussed previously, "33078: The Boston Celtics play their home games at TD Garden." The results seem similar to the previous attempt. Again this could be due to developer error.

Rank	Document ID	Similarity Score
1	List_of_role-playing ...	0.0093276934
2	Forplay	0.0073991531
3	LTD	0.0056983680
4	TD	0.0056983680
5	List_of_Bradford_Bulls_players	0.0053591947