Fact Checker
GIC

Paul Fährmann, 7006813,
Zun Wang, 6631108
Maria Carmela Dipinto, 6894020

Universität Paderborn

Abstract. The goal of the mini-project Fact-Checker is building a corpus-driven engine to check the facticity of the given facts (statements) from DBpedia. The Fact-Checker engine will generate a result file which stores the confidence values 0.0 (fact is false) or 1.0 (fact is true) for all given facts.

1 Introduction

The Fact Checker consists of three components as shown in figure 1.

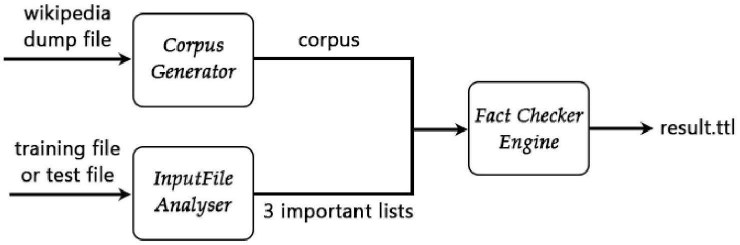


Fig. 1. Pipeline of Fact Checker

1. First component is the "Corpus Generator":
   The input of this component is the latest version of the English Wikipedia database dump file[1]. Through the Corpus Generator we can produce an offline corpus for the fact-checking. In section 2 the corpus generation will be explained further.

2. Second component is the "Input File Analyser":
   The job of the Input File Analyser is the extraction of the useful information from the statements in the training- or respectively the testfile. The extracted information is then stored in 3 lists: list of factID, list of statements' tokens and list of proper nouns of each statement. The details of this

component will be discussed in section 3.

3. Last component is the ”Fact Checker Engine”:
   The job of the Fact Checker is calculating whether the given statements are true or not. All results will be stored in the file ”result.ttl”. The algorithm of the Fact Checker Engine will be explained in detail in section 4.

All three components are implemented in Python.

2 Corpus Generation

Fact Checker is a corpus-driven engine, so the first task is generating a corpus. It is apparent, that the performance of a Fact Checker is highly dependent on a proper corpus.
If the corpus size is too large, then the Fact Checker would be inefficient, because the corpus contains probably too much junk or redundant information.
If the corpus size is too small, then it wouldn’t cover enough knowledge domains. Using a smaller corpus also requires a more sophisticated algorithm to generate correct results for each statements.
The internet can provide many ready-made corpora, but those are either very large (often not free) or just for a specific domain. None of them are suitable as corpus for our Fact Checker. Therefore after a discussion and inspiration by an article by KDnugget[2], we decided to use the Wikipedia’s database to build a proper corpus ourselves.
Most articles in Wikipedia have an abstract immediately beneath the title of that article. These abstracts contain already a lot of information about the corresponding article.
We guessed that taking the abstracts of the articles as the main content of our corpus would already be sufficient for solving the Fact Checking problem.

2.1 Use SAX-Parser to parse Wikipedia dump file

The first step of corpus generation is downloading a compressed version of the English Wikipedia dump file[1]. Wikipedia contains a lot of articles about a lot of topics, so it was not a surprise that after unzipping the package we got a 70GB XML dump file.
The Python library contains several kinds of XML parser. Many of those require to load the entire or partial XML file into memory before parsing it. This was obviously not possible for our 70GB XML file.
The SAX-Parser[3] however is an event-driven parser, which only reports the data of an event and discards all information once reported.
The memory efficiency was ideal for our problem of parsing our XML dump file. We define three event-callback methods as followed which are called when these events occur during parsing.

1. Element Start:
   If an opening tag (namely element start) is ⟨title⟩ or ⟨text⟩, we record the type('title' or 'text') of this opening tag. Both of the tags are very important for our corpus generation.
   Each Wikipedia article contains one tag pair ⟨title⟩ and ⟨/title⟩ and beneath it is another tag pair ⟨text⟩ and ⟨/text⟩. Between the opening tag ⟨title⟩ and the closing tag ⟨/title⟩ the title of a Wikipedia article is stored, and between the opening tag ⟨text⟩ and the closing tag ⟨/text⟩ the content (or text) of this Wikipedia article is stored. Both of these parts are needed for the corpus.

2. Text Node:
   If the current recorded tag-type is 'title' or 'text', we record all scanned text (content) into a list until we encounter the closing tag ⟨/title⟩ or ⟨/text⟩.

3. Element End:
   If a closing tag (namely element end) is ⟨/title⟩, we convert the recorded text between ⟨title⟩ and ⟨/title⟩ into a title-string.
   If a closing tag is ⟨/text⟩, we eliminate all useless elements from the record-list which stores all text between ⟨text⟩ and ⟨/text⟩, and combine all useful information into a text-string. At last, we write the title-string and the text-string into the corpus in the following form:

   1st. title-string
   text-string for the 1st. title
   ==========
   2nd. title-string
   text-string for the 2nd. title
   ==========
   3rd.title-string
   ......

   A line filled just with "==========" separates two articles.
   Each corpus file gets filled with these lines of text until it reaches the size of around 45MB. This was important for us to be able to upload the corpus to GitHub.

2.2 Extraction of useful Information

As mentioned above, our corpus doesn't need to include the entire article. We take the abstract of each article as the main content of our corpus. Directly beneath the abstract is a catalog for the corresponding article. The text of the first item in the catalog often contains important information. So we also take it into the corpus. We ignore the rest as it would make our corpus way too big without giving us much more information.
The SAX-Parser scans the text between opening tag ⟨title⟩ and closing tag ⟨/title⟩ line by line. Each line gets stored as an element in a record list, from

which the important information will be extracted. The process of extracting useful information is written down in the pseudo code – Algorithm 1: Clean_Raw_Text. Although the information extracted with Clean_Raw_Text is not so clean, the corpus which is built with the information is good enough for fact checking.

Algorithm 1 Clean_Raw_Text

```
Input: Record-List of a Wikipedia text
Output: Text-String that stores all extracted useful information
Find the first element S in Record-List whose first word is in the form of ””’word””’(bold and italic) or in form of ”’word”’ (bold).    ▷ Find the first line of article abstract.
Delete all elements in front of this element S except S.
Find the second element E in the list which is in the form of ==subtitle==, if it exists.    ▷ Find the subtitle of the second item in the article catalog
Delete all elements behind the element E including E.
Combine all elements in the Record-List to Text-String.
Delete all parts in form of {{reference}} from the Text-String, using regular expression.
Delete all special symbols from Text-String except for letter and number, regular expression.
Delete all white-space from Text-String except single space, using regular expression.
return Text-String
```

2.3 Generation with Multiprocess

Finding the above described way to extract and clean the right texts from the XML-file for the fact checker took several iterations.
It was therefore important to optimize that generation process as the generation of the whole corpus took more than one day to finish when it ran on only one processor.
We optimized this corpus generation by using multiprocessing for Clean_Raw_Text and saving the result to the corpus-files.
Instead of scheduling all the articles one after another on one process, the main process gathers the Record-List of one whole article and puts it onto a free process. This process then extracts the important parts from the Record-List by executing Clean_Raw_Text. It then writes the result to the corpus-files.
If no process is free at that time, the main process waits with the Record-List for one to finish its task.
This change sped up the corpus-generation enough for us to comfortably find the correct extracting and cleaning methods.

3 Analysis of Input File

We are given two input files: SNLP2019_training.tsv and SNLP2019_test.tsv. The statements to be queried are stored in these two files. The task of this step is reading the input file line by line, decomposing each line and storing it into three lists. These three lists are very important for generating the result file.

1. IDlist: list of FactID
   The first part of each line in the input file is the FactID. We store all FactIDs in a list, that is later used for generating the result file.

2. tokenslist: list of statements' tokens
   The second part of each line in the input file is the statement, for which we need to find the truth value. We handle each statement as following steps:
   - Step 1: Eliminate all symbols except for letter and number.
   - Step 2: Tokenize the statement and store all tokens in a list *l*.
   - Step 3: Eliminate the empty element from the list *l*.
   - Step 4: Store *l* in tokenslist.

3. nameslist: list of proper nouns in each statement
   We find that in each statement both the subject and the object are proper nouns, e.g.: name of a person, name of a film, name of a country, etc. These names are important information for judging the truth of a statement. The feature that distinguishes a proper noun from any other word is its capitalization. We utilize this characteristic to generate the nameslist:

```
create the nameslist
for each $l$ in tokenslist do
    create a list $n$
    for each token in the list $l$ do
        if token's first letter is upper case or token == 'von' then
            Convert the token into lower case
            Add the token into the list $n$
        end if
    end for
    add the list $n$ into the nameslist
end for
return nameslist
```

4 Check Facts

The remaining work is to check the truth of the list of statements.
Our approach is very simple, it simulates the process of searching keywords in

a search engine. In this mini-project the keywords are the proper nouns of each statement. We search in the created corpus for the keywords.
An example:

Statement: ”Camp Rock stars Nick Jonas.”

As mentioned in the 3rd section, we tokenize this statement to get a list L:

$$L = [\text{'Camp', 'Rock', 'stars', 'Nick', 'Jonas'}]$$

Then we extract the proper nouns from L and store them in lowercase in a list PN:

$$PN = [\text{'camp', 'rock', 'nick', 'jonas'}]$$

We convert L into lower case:

$$L = [\text{'camp', 'rock', 'stars', 'nick', 'jonas'}]$$

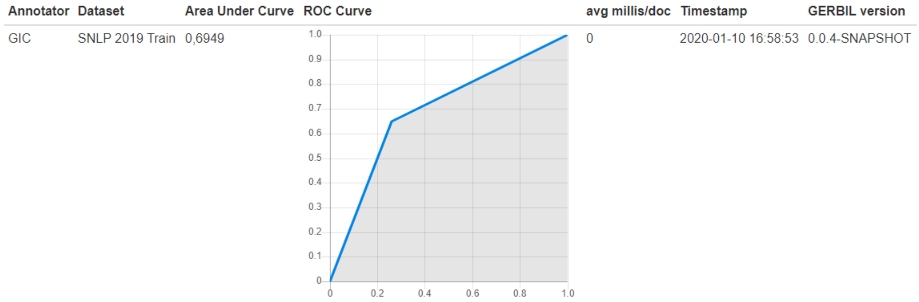
We search in the corpus. If there’s an article title whose tokenized tokens in lower case are all included in the list L, we go through that articles text.
We preprocess the text to get a text-token-list:

1. Eliminate the symbols from the text except letter and number;
2. Utilize the white spaces to split the text in tokens and store them into a text-token-list;
3. Delete empty elements from the text-token-list;
4. Convert all tokens into lowercase;

At last we check whether the proper noun tokens in PN are all included in the text-token-list. If yes, we assert that the corresponding statements is true: result = ’1.0’.
If we couldn’t find such an article(title+text) in the corpus, we assert that the corresponding statements is false: result = ’0.0’.
For our example, we find at least two articles in the corpus, one with the article ”Camp Rock” and another with the article ”Nick Jonas”. Because [’camp’, ’rock’] and [’nick’, ’jonas’] are sublists of L = [’camp’, ’rock’, ’stars’, ’nick’, ’jonas’]. We preprocess both of the article texts to get the text-token-lists.
We check whether the proper noun list PN = [’camp’, ’rock’, ’nick’, ’jonas’] is sublist of the text-token-lists. If so, the result = ’1.0’ for the statement ”Camp Rock stars Nick Jonas.”. Otherwise, the result = ’0.0’.
For this example our Fact Checker Engine will find the correct result in both the articles.
The results of all statements are stored in the results-list. With this results-list and IDlist the Fact Checker Engine will generate the result file.

5 Result

We use our Fact Checker Engine to generate two results-files:

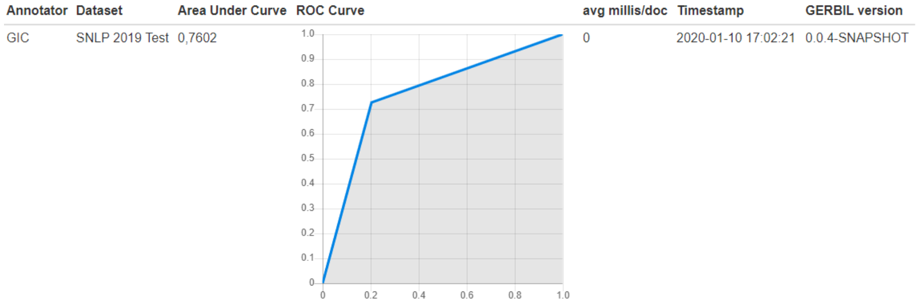


Annotator	Dataset	Area Under Curve	ROC Curve	avg millis/doc	Timestamp	GERBIL version
GIC	SNLP 2019 Train	0,6949		0	2020-01-10 16:58:53	0.0.4-SNAPSHOT

Fig. 2. AUC for the results-file: training_result.ttl

- The results-file 'training_result.ttl' is for the input file 'SNLP2019_training.tsv': The AUC[4] of these results is as shown as in the Fig.2.

- The results-file 'result.ttl' is for the input file 'SNLP2019_test.tsv': The AUC[4] of these results is as shown as in the Fig.3.



Annotator	Dataset	Area Under Curve	ROC Curve	avg millis/doc	Timestamp	GERBIL version
GIC	SNLP 2019 Test	0,7602		0	2020-01-10 17:02:21	0.0.4-SNAPSHOT

Fig. 3. AUC for the results-file: training_result.ttl

5.1 Discussion

In section 4 we already discussed a case for which our algorithm generates the correct truth value. Our approach is very simple, therefor it's easy to find examples with wrong evaluations for our system: comparing our results in the 'training_result.ttl' with 'SNLP2019_training.tsv' file, we following statement with ID 3812179:

"Franklin D. Roosevelt's subsidiary is Warm Springs, Georgia"

This statement is actually False (0.0) as specified in the training-file while our system generates True (1.0), as found in our result-file.
This is a False Positive example. The error is explained by the simplicity of the approach that our system uses: we consider just the proper nouns in the statement and no other words.
In this case one of the lowercase words (i.e. "subsidiary") was very important for checking the statement, as it specified the exact information of that statement and connected the proper nouns meaningful.
So one improvement to our system could be considering not only the proper nouns but also the other words, that specify the meaning of the statement, to check the truth value.
Another failure-case is in the statement with ID 3213978:

"Cam Reddish's team is Atlanta Hawks"

Here we can see a False Negative example, a statement that is actually True (1.0) but that's classified as False (0.0) by our system.
This comes from the difference of how the proper nouns are written in the statement and how they are written in our corpus.
The same name "Cam Reddish" in the statement, is found in the Wikipedia text as "Cameron Elijah Reddish".
Our system is not able to consider the statement as True, because it's not able to find the proper noun "Cam" in the text, even though "Cam" is found in the title of the article.
To improve the system for this case, we could add the title of the article to the text-token-list that is specified in section 4.
Another source for False Negatives could be the size of our corpus, as we don't consider all the text of the articles. However the text could include important information for a statement.
One strength of our approach is also the simplicity, because we can understand easily why the system thinks a statement is true or false. Also our algorithm is fast and can be used as a base to build a more sophisticated algorithm.

References

1. English Wikipedia database dump file, https://dumps.wikimedia.org/enwiki/latest/
2. KDnuggets, Building a Wikipedia Text Corpus for Natural Language Processing, https://www.kdnuggets.com/2017/11/building-wikipedia-text-corpus-nlp.html
3. Simple API for XML(SAX), https://en.wikipedia.org/wiki/Simple_API_for_XML
4. AUC Calculator, http://swc2017.aksw.org/gerbil/config