

Virginia Tech
Department of Computer Science
Blacksburg, VA 24061

CS 4624 - Multimedia, Hypertext, and Information Access
Spring 2019

Conversation Facts

Jonathon Marks
Nathan Miller
Christian Kang

Client: Saurabh Chakravarty
Professor: Edward Fox
May 8, 2019

Table of Contents

Table of Contents	1
List of Figures	3
List of Tables	3
Executive Summary	4
Introduction	5
Requirements	7
Design	9
Implementation	10
Parsing	10
Preprocessing	11
Rule Based Filtering	11
The Apostrophe Problem	12
The Coreferencing Problem	12
Generating Knowledge Graphs	14
Initial Knowledge Graphs	14
Chunking/Chinking Knowledge Graphs	15
Incorporating Dependency Parsing	16
Dependency Parsing Knowledge Graphs	17
Identifying referenced location	19
Testing/Evaluation Assessment	20
Defining Random Baseline	20
spaCy Baseline	20
Our Method	21
Overall Results	22
User's Manual	23
Dependencies and Project Setup	23

Python Package Installation	23
GraphViz Installation	23
Github	23
Running the Project	24
Front End Product	24
Running Specific Files	26
Docker	27
Developer's Manual	28
File Inventory	28
Dependencies and Project Setup	29
Refer to the previous section for installation requirements and instructions.	29
Project Overview	29
Front End Product	29
Pipeline	29
Lessons	31
Verifying library abilities is important	31
Evaluation of NLP techniques is difficult	31
Not all NLP methods are useful for every problem	31
Processing internet text conversations is a challenge	32
Off the shelf libraries don't work for all cases	32
Future Work	34
Acknowledgements	34
References	35
Appendices	36
A.1 NLP Parts of Speech	36
A.2 spaCy Syntactic Dependency Labels	37
A.3 Meeting notes	39

List of Figures

Figure 1: Entity-Relation-Entity (ERE) Triple Example	5
Figure 2: Part of Speech (POS) Tagging Example	6
Figure 3: Design Flow	9
Figure 4: Partially formatted conversation data	10
Figure 5: Partially formatted summary data	10
Figure 6: Examples of short sentences without obvious EREs	11
Figure 7: Coreference Example (Conversation)	13
Figure 8: Coreference Example (Summary)	14
Figure 9: Sentences split on first verb	14
Figure 10: Knowledge graph from splitting on first verb	14
Figure 11: A sentence chunked into noun phrases	15
Figure 12: Tag pattern used for chunking/chinking full verb phrases	15
Figure 13: Knowledge graph from chunking/chinking	16
Figure 14: AllenNLP dependency parse of a sentence	16
Figure 15: Chunking/chinking knowledge graph and knowledge graph using dependency parsing	17
Figure 16: Sample dependency parse tree using spaCy library	18
Figure 17: Sample sentence root with negation dependent	18
Figure 18: Results of summary sentences to dialog matching	20
Figure 19: Examples of summarizers using S1 tag	21
Figure 20: Overview of the front end html page	24
Figure 21: Front end html conversation selection	25
Figure 22: Front end html conversation text	25
Figure 23: Front end html summary texts	25
Figure 24: Docker container	27
Figure 25: Example dictionary in data.json	30
Figure 26: Entry object created in parse.py	30

List of Tables

Table 1: List of executable files	26
Table 2: List of project files	28
Table 3: NLTK Parts of Speech	36
Table 4: spaCy dependency relations	37
Table 5: Meetings notes and goals	39

Executive Summary

The goal of the Conversation Facts project is to be able to take a summary of a conversation and link it back to where it occurs in the conversation dialogue. We used the *Argumentative Dialogue Summary Corpus: Version 1* [1] from Natural Language and Dialog Systems as our dataset for this project. This project was created in Python due to its natural language processing libraries which include spaCy [5] and the Natural Language Toolkit (NLTK) [8] libraries. These two contained the methods and techniques used in the project to parse the data and process it into the parts of speech for us to work with.

Our general method of approach for this project was to create knowledge graphs of the summaries and the conversation dialogues. This way, we could connect the two based on the entity-relation-entity (ERE) triples. We can then compare the summary triple which would point us back to a corresponding conversation triple. This will link back to the section in the dialogue text that the summary is referencing.

Upon completion of the project, we have found that our methods outperforms naïve implementations of simply running our data through industry standard software, but there are still many things that could be improved to get better results. Our program focuses on utilizing natural language processing techniques, but we believe that machine learning could be applied to the data set in order to increase accuracy.

This report explains the requirements set for the team to accomplish, the overall design of the project, the implementation of said design, and evaluation of results. It also includes a User's Manual and Developer's Manual to help illustrate how to either run the source code or continue development on the project. Finally, we describe the lessons learned throughout completing the project and list the resources used.

Introduction

The Conversation Facts project was proposed by Saurabh Chakravarty as a way to help him in his research in natural language processing. The purpose of the project is to be able to find the parts of a conversation that correspond to a summary of that conversation. This would help in cases where there are long dialogues between two people and somebody wants to see where in the conversation a fact comes from. For example, if somebody came across a summary of an interview between two people, it would be helpful to find exactly what part of the interview a statement from the summary comes from.

Saurabh became both our team's client and mentor as he set goals for us to accomplish and guided us through the project. Normally, a project like this would include some sort of machine learning; however, none of the team had any experience with machine learning which would take too long to learn in one semester. For this reason, Saurabh had us focus on tackling the project using natural language processing techniques. He pointed us towards certain resources that would teach us along the way about natural language processing since none of us had much experience in that field either. An example of some of the resources were the online lectures given by Stanford professor Christopher Manning [2]. In these lectures, Manning gives a series of videos which describes the basic techniques of natural language processing which are essential to this project.

There are several fundamental terms that should be known to comprehend this project. To begin, we will define the required vocabulary of Knowledge Graphs or KGs. The first thing to understand are the building blocks of KGs: Entity-relation-entity (ERE) triples. These are a model to define a relation between two objects (see Figure 1). EREs take the two entities as nodes and connects them by a relation which is represented as an edge. KGs are essentially a network of these entity nodes that are connected through the relation edges.

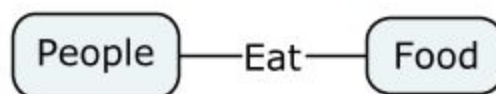


Figure 1: Entity-Relation-Entity (ERE) Triple Example

To create these EREs, it is necessary to parse through the text to understand what kinds of words are involved in each sentence. This is where Part of Speech (POS) Tagging becomes useful. POS tagging is the process of marking up the words in a corpus with their corresponding part of speech (see Figure 2). With the words marked, it is possible to understand which ones should be entity nodes and which ones should be relations between the entities. There are many parts of speech that will not be explicitly discussed in this section; however, the main tags we are concerned with are nouns and verbs. With these, we can usually create ERE triples using nouns

as entities and verbs as the relationships between those entities. (See the Appendix for a full list of parts of speech.)

```

My ADJ
conduct NOUN
is VERB
always ADV
professional ADJ

I PRON
conduct VERB
myself PRON
in ADP
a DET
professional ADJ
manner NOUN

```

Figure 2: Part of Speech (POS) Tagging Example

POS tagging is essential to creating the EREs, but some words in sentences are not always necessary. For example, words like “a”, “but”, and “or” are not usually fundamental in order to understand sentences. These are what can be considered “stop words” and are usually filtered out during the preprocessing step called stop word removal. This can be helpful so that people do not have to worry about including these unnecessary words in EREs and the final knowledge graph. This being said, sometimes stop words are helpful in understanding a sentence. The word “not” is a critical segment of a sentence as it negates some statement which would otherwise be ignored by stop word removal. This can have huge effects on the meaning of a sentence which may lead to wrong interpretations.

Now that we have covered part of speech tagging, we can use this knowledge to organize words into groups and phrases. We use a process called chunking to “chunk” parts of speech together. Generally, we do this by selecting a regex of parts of speech that create a phrase. This way, we can define a noun phrase as not just a noun but also include any adjective that describes the noun or any determiner that precedes the noun. This helps when there are multiple parts of speech that you want to include as an entity or relation. Additionally, there is a similar term called chunking which is essentially the opposite of chunking. You might think of it as a “chink” in armor where a piece has been taken out. This is used when you create a phrase with chunking and want to take out certain things from that phrase. It is especially helpful when you want to make further separations within chunked phrases.

Lastly, we will cover the term lemmatization. This is the process of grouping different forms of a word into its base structure. This reduces the different tense versions of a word to its dictionary form. An example would be reducing the words playing, played, and plays down to the base play. This helps when trying to create knowledge graphs since some concepts will revolve around the same idea but may have different terms to describe it. You would not want two different ERE triples which define the same exact relation. Instead, you would want to lemmatize it down into its base term so that you do not repeat similar ideas.

This vocabulary is essential to understand in order to comprehend this paper as they will be used extensively to describe this project. The rest of the paper will cover the requirements set out by Saurabh, the design, implementation, and evaluation of the project, a user and developer manual for the purpose of informing someone on how to execute it or how to continue work on it, and then the lessons learned throughout this project.

Requirements

The ultimate goal of the Conversation Facts project was to research ways to link summary sentences back to the section of the conversation where it is referenced. Since the goal of the project was research based and did not focus on finding a definitive answer, the requirements were set to guide the team towards smaller objectives. Saurabh set the series of milestones for the team to accomplish throughout the semester in a way such that we would learn piece by piece the information that was necessary to complete the project. These requirements were more of a mapping of the general project structure rather than true deliverables of products to be complete.

Milestones:

1. Choose a suitable dataset
2. Understand and be able to hand-draw Knowledge Graphs
3. Understand and implement sentence chunking, lemmatization, and stop word removal.
4. Write code to create entity triples of summaries
5. Write code to create entity triples of conversations
6. Compare and connect the two sets of ERE triples

The first and perhaps one of the most important things to consider was choosing a suitable dataset. There were several corpora available online that contained conversations between two people, but finding the one that would be most fitting proved to be a significant part of the project. We wanted to find a dataset that was composed of plaintext as we would be able to manually see the effects of our processing. There were many that were already processed into parts of speech or into dependency trees which we decided not to use. We also needed to have a corpus that came with a summary. With these criteria, we decided to use the dataset *Argumentative Dialogue Summary Corpus: Version 1* from Natural Language and Dialog Systems [1].

The next step was to understand knowledge graphs or KGs. Since KGs are the main subject for this project, Saurabh wanted us to understand them by generating the graphs by hand. This would get us to think about the way that we as humans decipher and comprehend language. Once we got through this phase, we would have learned first hand the problems and difficulties of creating KGs before we tried to generate them with code. We will go through some of the difficulties experienced and the knowledge gained on this later in the Lessons section of the paper.

Another large portion of this project was being able to understand the terms and techniques used in natural language processing to generate knowledge graphs. Chunking, chunking, stop word removal, and lemmatization were a few of the techniques that we researched in order to try out different ways of processing our data. The processing would make it easier to create the ERE triples from the data. However, not all of these methods are applicable to our specific dataset. This will also be discussed in the Lessons section of the paper.

In order to continue with the project, we had to put our understandings of the techniques we had researched into practice. We did this by using natural language processing Python

libraries which provided the tools to help develop code. Other methods that were not readily available with libraries were implemented by the group. As we wrote the code, we found that the sentence structure of the conversations and summaries were very different. Thus we have had to go through several variations of combining these techniques in order to successfully create ERE triples.

Finally, the last milestone to complete was to find ways to compare and connect the two sets of ERE triples. This was done with several techniques including term frequency - inverse document frequency (tf-idf) and keyword matching. With this completed, we would be able to produce results on how well our methods worked.

Design

Due to the nature of this project, the design was heavily dependent on the dataset that was chosen. The main aspect to find in the dataset was to find one that contained conversations between two humans that also includes a summary of the conversations. However, different datasets provide different contexts for the type of information that they hold. This includes a range of different corpora such as question-answer type interviews, deliberation over a topic to achieve a common goal, and debates over an argument. Our dataset, *Argumentative Dialogue Summary Corpus: Version 1* [1] contains 45 conversations on gay marriage with five summaries accompanying each conversation. These conversations took place electronically in the form of online forum discussions. Each of the five summaries that are paired with conversations were generated by people where they were asked to create summaries in their own words. Since the corpus follows an opinionated debate structure between two people, it changes the way that dialogues take place. There is not a simple question and answer which makes for simple knowledge graphs. Instead, there are more vague concepts being argued which creates challenges for how we approach the problem.

The basic method of generating knowledge graphs was suggested by Saurabh as a good way to tackle the problem. The way we approached the project to accomplish this was to first parse our dataset into objects that we could work with. Next, we would use preprocessing techniques on the organized data to clean up some of the poorly formatted data. Then we could take the cleaned data and try to build EREs out of it. Finally, the two sets of EREs would be compared to see if we could come up with a match with summary sentences and conversation dialogues.

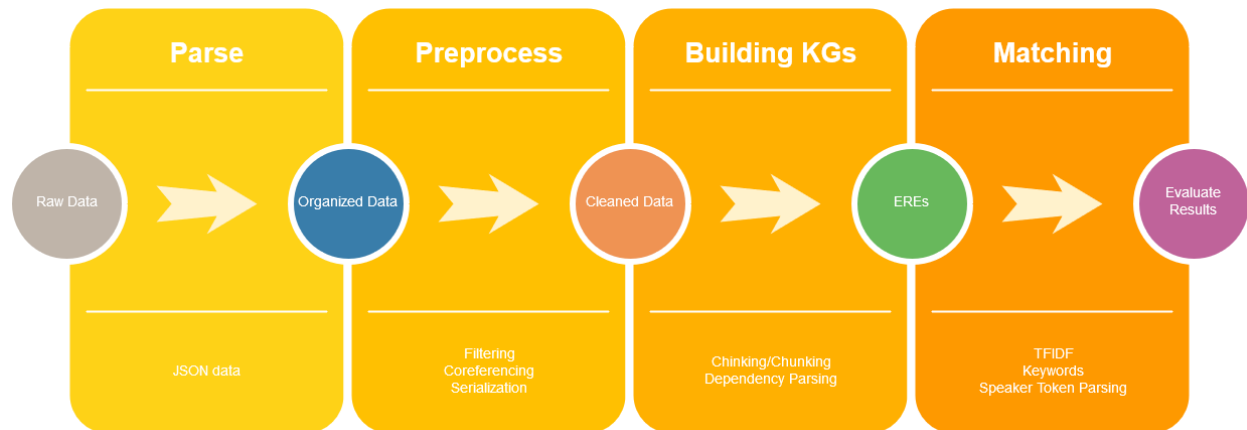


Figure 3: Design Flow

Implementation

This section describes our general approach towards implementing our chosen design. In it we discuss work we have done in order to reach different milestones. We also discuss how our design and strategy has evolved throughout the project.

Parsing

The first challenge of the project was parsing the data into a form that we could actually use. The initial data for each conversation summary pair is provided in JSON. However the entirety of the conversation and all 5 summaries are each given as a single value. In order to differentiate the speakers they are marked as “S1:##” or “S2:##” where the pound signs are replaced with the number of times they have spoken, as shown in Figure 4.

```
S1:1- Another problem with the study is that most ( or at least more ) |
S2:1- S1 you don't know how they conducted the study. You don't know wh
S1:2- Yes, that's the point. I'd prefer to know the specifics. Question:
S2:2- So then question, don't just state that the study has a problem.
S1:3- Er... that's why I pointed out the specific things I saw as a probl
S2:3- No more than usual. :) You, however, are stating that some is a pi
```

Figure 4: Partially formatted conversation data

Note that in the actual data each speaker does not begin at a new line so the “S#:##” tags must be used to differentiate text. We split these conversations on the “S#:##” tags using Python’s regex so that each conversation could be represented as a list of alternating speakers. In addition we stripped the conversations of almost all special characters to allow for easier creation of knowledge graphs. Notice for example the “:)” text in S2:3 of the conversation in Figure 4.

The summaries on the other hand were separated by a number of newlines, dashes, and a label in the form of “D#”.

```
-----\n D0\n -----\n
S1 and S2 are discussing a seemingly pending law that
\n \n \n -----\n D1\n -----\n
S1 believes that religious freedom allows for people
\n \n -----\n D2\n -----\n
The issues here appears to stem from an employee citi
\r\n \n \n \n -----\n D3\n -----\n
S1 and S2 are discussing equality and natural rights.
\n \n \n -----\n D4\n -----\n
S1 and S2 are discussing gay marriage. The discussio
```

Figure 5: Partially formatted summary data

Again in this instance we were able to split apart the summaries using Python's regex and remove the numerous newlines. The summaries then went through the same removal of special characters that the conversations did. While emojis, "emphasis" quotes, and other special characters were less numerous in the summaries they commonly had several newline characters in between different ideas.

Preprocessing

The second step of our implementation is our preprocessing step. This filters out more advanced extraneous text than just special characters from the parsing implementation. In addition it resolves ambiguous pronouns from coreferencing and serializes the data object so that parsing and preprocessing only has to happen a single time.

Rule Based Filtering

Rule based filtering is used to remove text that reduces the accuracy of later methods such as dependency parsing and chunking/chunking. This stage removes text between parentheses, short sentences under a given length, and replaces common words that use apostrophes.

Short sentences were particularly difficult to parse and difficult to match. Because of their size they also rarely contained relevant information. Short sentences were often simply exclamations, insults, or other text that had been poorly split due to extraneous periods or mistakes making the database.

```
And \" vilify \"? Good grief! Where did you get that from?  
S2:3- Here you go :  
S2:1- LOL. :p OK, if you say so. Waxy
```

Figure 6 : Examples of short sentences without obvious EREs

Text within parentheses causes a similar problem. Much like the short sentences the text contained within is often not relevant to the general ideas being expressed. In addition by removing the parentheses by themselves the structure of the sentence is changed such that it makes parsing with NLP tools more difficult.

By removing this text we are able to greatly reduce the issues that they caused. More details on issues from apostrophes are listed below.

The Apostrophe Problem

Throughout the project we dealt with a preprocessing issue caused by apostrophes. NLTK/spaCy often split words containing apostrophes into separate tokens. This caused later issues when using Graphviz as there was tokens containing only a single apostrophe or “ ‘s”. This was often detrimental to the building of knowledge graphs because often times splitting on the apostrophes removed or changed the meaning of the word. Several techniques were tried including:

1. Removing all apostrophes during preprocessing
This fixes the issues of stray apostrophes causing problems in Graphviz. Doing this, however, also removes context and can change the word’s meaning so we ultimately did not use this solution.
2. Changing words with apostrophes
Changing each word to synonymous words that don’t contain apostrophes was proposed by Saurabh. This was implemented for many of the more common words containing apostrophe. However this does not work for all tokens as some words can-not be easily changed by removing the apostrophe. This, like indicating possession, would require sentence rewrites. So while this method is used in the final draft it is not able to handle all cases.
3. Editing the Graphviz format
This is the second half of our final solution. Ultimately we were able to change the way we were using Graphviz so that apostrophes would not break the syntax of the .dot files. This combined with our replace dictionary allowed us to keep the overall context for the majority of words while not causing any breaking bugs for less frequent, less replaceable words.

The Coreferencing Problem

One of the main difficulties in trying to analyze any conversation is the ambiguity of when an entity refers to something previously stated. When speakers mention themselves or the other party, it is almost always in the form of pronouns like “I”, “me”, and “you.” These pronouns pose a problem as they do not provide any context to whom is speaking when each sentence is looked at individually. This does not help since it then creates ambiguity while generating ERE triples. If every relation is “I” or “you”, it is impossible to decide which speaker the relation originally pertains to.

We had found that not only are there problems with ambiguity in the conversations, but also in the summaries. The dataset tries to keep the genders of speakers unknown by using gender neutral pronouns within the summaries. This can make it even harder to resolve ambiguity than in conversations for several reasons. One reason is that the gender neutrality can cause confusion as to which speaker a summary is referring to. If a summary sentence talks about both of the speakers and the following sentence uses a pronoun or a phrase like “this person,” it is difficult to determine whom the pronoun refers to. Another reason is that the summarizer will often use the gender neutral pronoun “they” which makes it confusing as to whether it is referencing a speaker or some other group of people mentioned in the previous sentence.

To address this problem, we used coreference resolution. This is the practice of finding expressions that refer to the same entity. Originally we had tried to use off-the-shelf software such as spaCy's NeuralCoref library and AllenNLP's coreference model, but neither of these worked for our dataset which will be discussed in the Lessons section of the paper. This made us realize that it will be difficult to resolve anything more than the speakers. Focusing on only resolving pronouns that refer to speakers, we had to come up with our own techniques. As mentioned previously, both the conversations and summaries contained pronouns that had to be resolved. However, the conversations' and summaries' usage of pronouns differed so much that we had to design two separate methods for coreferencing.

The first method pertains to coreferencing the conversations. Within a conversation, two people will be speaking. The first speaker will say a few sentences and then the second speaker will reply. This strict turn taking makes it easier to resolve pronouns that refer to either of the speakers. Firstly, we know that the pronouns will usually be limited to words like "I", "me", or "you" so we would only have to look for those within a dialogue. Second, we can look to see who is speaking and resolve the pronouns based on whose turn it is. If Speaker 1 is talking and says "you," we can often assume that the "you" refers to Speaker 2. In the other case if Speaker 1 says "I" we can guess that they are referring to themselves. With this, we were able to design a method to replace the pronouns by mainly looking at the parity of whose turn it is.

S1: Evidently you didn't read the links I gave you.	S1: Evidently S2 didn't read the links S1 gave S2.
S2: I clearly did, I suspect you may have but didn't understand.	S2: S2 clearly did, S2 suspect S1 may have but didn't understand.

Figure 7: Coreference Example (Conversation)

The second method we implemented tried to resolve pronouns in the summaries. This was more difficult than the conversations due to the problems mentioned previously. We cannot blindly replace pronouns since we do not know whether the summarizer is using "they" or "he/she" or phrases like "this person." We cannot make an accurate guess as to which speaker a summary refers to, based upon which sentence number we are on. Therefore, we had to design an algorithm that would take into account the dependency labels that are produced for each sentence. More than likely, if a summarizer references back to a previous sentence, then they are usually referring back to the subject of the sentence. Our approach looks at certain label patterns, specifically the 'nsubj' tag, to find the subject of a sentence and store it if it is not a pronoun. Then if a pronoun or phrase refers back to something, it will possibly be the stored subject. This requires running each sentence through spaCy to create the dependency labels which also means that our method is relying on the accuracy of the dependency parser. Some sentences will not

follow this structure which makes this method less accurate than the one designed for conversations.

<p>1 S1 argues that most people in the country might identify as Christian and that questioning a study is a very appropriate response to one that vilifies a group of people.</p> <p>2 They believe that there is a lot of attention given to trying to make Christianity look bad through generalization, irrelevant studies and misinterpretations.</p>	<p>1 S1 argues that most people in the country might identify as Christian and that questioning a study is a very appropriate response to one that vilifies a group of people.</p> <p>2 S1 believe that there is a lot of attention given to trying to make Christianity look bad through generalization, irrelevant studies and misinterpretations.</p>
--	--

Figure 8: Coreference Example (Summary)

Generating Knowledge Graphs

The third step of our process is to split the text into ERE pairs that we can use to build knowledge graphs. This process went through many different iterations and utilized many different techniques throughout. The techniques and development process is detailed below along with information about our final implementation.

Initial Knowledge Graphs

The initial design of the knowledge graph was based around splitting each summary on the first verb found. This would create a graph where each edge would be a verb and in theory would link a noun phrase to a more complex idea phrase. Examples of sentences split in this manner can be seen in Figure 9.

```

s1_ -> argue -> people_country_identify_christian_question_study_appropriate_response_vilify_group_people_
s1_ -> believe -> lot_attention_give_try_christianity_look_bad_generalization_irrelevant_study_misinterpretation_
s1_ -> believe -> divorce_rate_christians_bearing_gay_marriage_amendment_
s2_ -> believe -> s1_make_objection_decide_factual_evidence_
s2_ -> believe -> objection_hypothetical_actual_prove_one_
s2_ -> think -> dismiss_result_check_validity_reactionary_
s2_ -> think -> relevant_long_people_protect_sanctity_marriage_
    
```

Figure 9: Sentences split on first verb

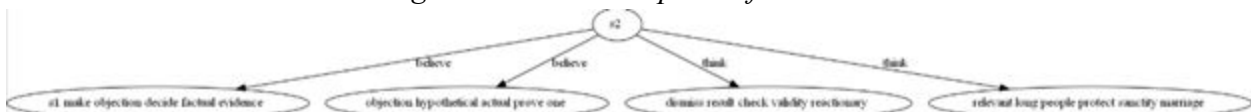


Figure 10: Knowledge graph from splitting on first verb

However this method both had issues generating knowledge graphs for the conversations and also generated unhelpful knowledge graphs for the summaries. Most ERE triples in the summary took the form of “S1/S2 -> thinks/believes/argues -> <complete sentence>”. This is not particularly helpful for our goal as it at best tags each speaker to a sentence. It does not tag each speaker to an idea/topic or ideas and topics to each other. Since our ultimate goal is to find where in the conversation a summary topic came from this method was not helpful.

Chunking/Chinking Knowledge Graphs

Our next method was to generate knowledge graphs using chunking and chinking to more intelligently split the sentences into noun phrases and verb phrases. Chunking is the process of grouping together adjacent tokens into larger “chunks” based on their parts of speech. Chinking is the process of removing a series of tokens from a chunk. It can also be thought of as defining what parts of speech shouldn’t be put into a chunk. In our case we can define noun phrase chunks broken up by verb phrase chinks. This would allow us to generate more than a single ERE triple from each sentence and also improve the quality of the ERE triples.



Figure 11: A sentence chunked into noun phrases.

Initially we just used chunking for noun phrases and chinking for individual verbs. This worked much better than our initial implementation, however it still had some issues. Namely when multiple verbs were adjacent to each other we would have to reconnect them to create an edge. In addition, several parts of speech such as determiners, to and its variations, and particles made more sense attached to the verb making up the edge. This led to us not only chinking out individual verbs but instead custom verb phrases.

```
tag_pattern = ""NP: {<.*>}
              } (<DT|RB.*|TO>?<VB.*>+<DT|RB.*|TO>?)+{
              VP: { (<DT|RB.*|TO>?<VB.*>+<DT|RB.*|TO>?)+ }
              ""
```

Figure 12: Tag pattern used for chinking/chunking full verb phrases

This caused a more free flowing structure that didn’t drop as many words or create artificially long entities in ERE triples. An example of this can be seen in Figure 13 where “divorce rate of Christians” is connected by “has no” instead of just “has”.

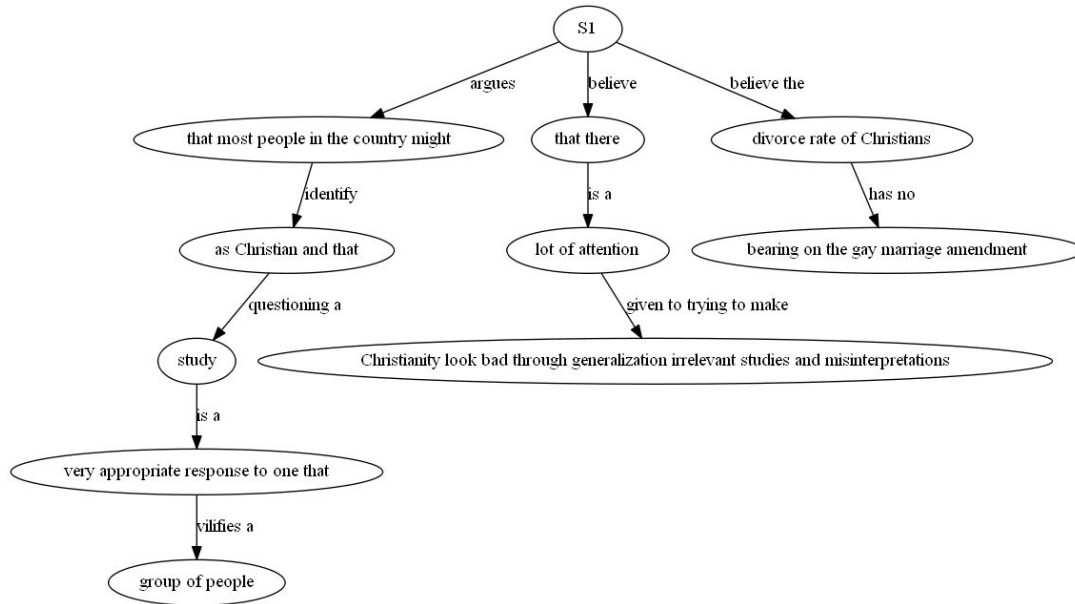


Figure 13: Knowledge graph from chunking/chinking noun/verb phrases

This has helped make our knowledge graphs much more legible. However there were still issues such as the extraneous edges of S1/S2 believes, thinks, argues. Also our graphs were still fairly thin. Rather than being an entanglement of related nodes they were closer to a tree structure with each path out of the root being a single sentence.

Incorporating Dependency Parsing

In order to resolve these issues we began looking to dependency parsing rather than the chunking/chinking design. This way we were able to more intelligently build ERE triples and prune out ones that did not provide additional context. See Figure 14 and Figure 15.

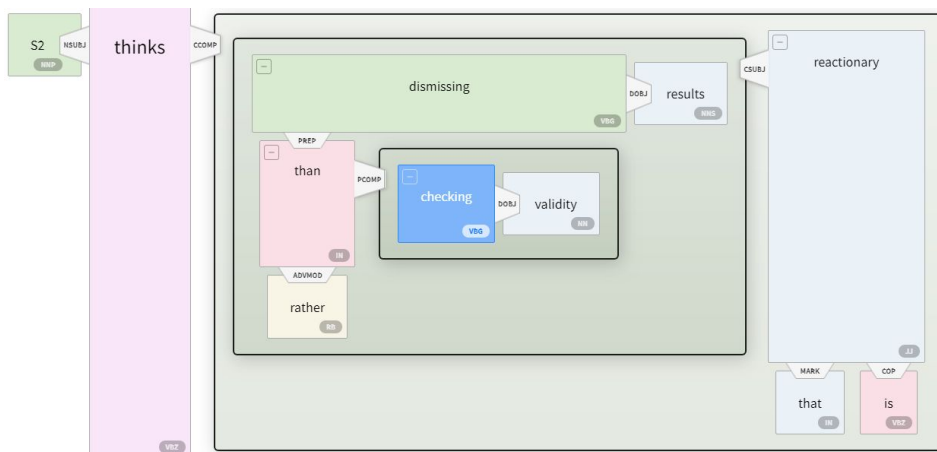


Figure 14: AllenNLP dependency parse of a sentence

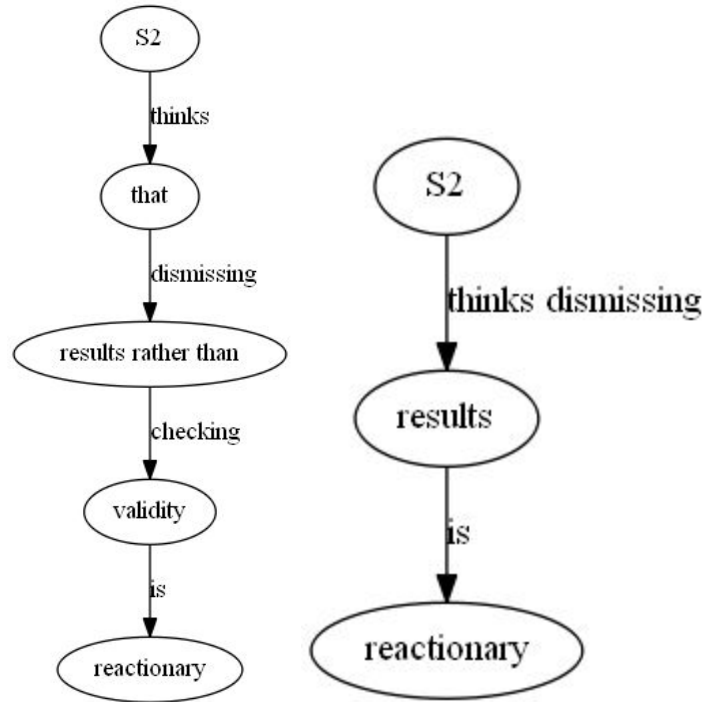


Figure 15: Chunking/chinking knowledge graph and knowledge graph using dependency parsing

As shown above by using dependency parsing we can reduce the number of extraneous nodes and edges in our graph. In addition this can improve our graph by allowing ERE triples to exist without needing additional context from other nodes. In this example only words with dependents or connecting words are used to generate the knowledge graph. This can be seen by reading the top line of Figure 14. The verbs ‘thinks’ and ‘dismissing’ are then merged in accordance to our rules on verb and noun phrases. This creates a much smaller but much more clear graph.

Dependency Parsing Knowledge Graphs

A dependency parser analyzes the grammatical structure of a sentence and establishes relationships between “head” words and words which modify those heads. The dependency parser establishes a root word and builds a tree structure from there containing the rest of the sentence. For a list of dependency tree relations look to Appendix A.2. Figure 16 shows a sample sentence along with its dependency tree.

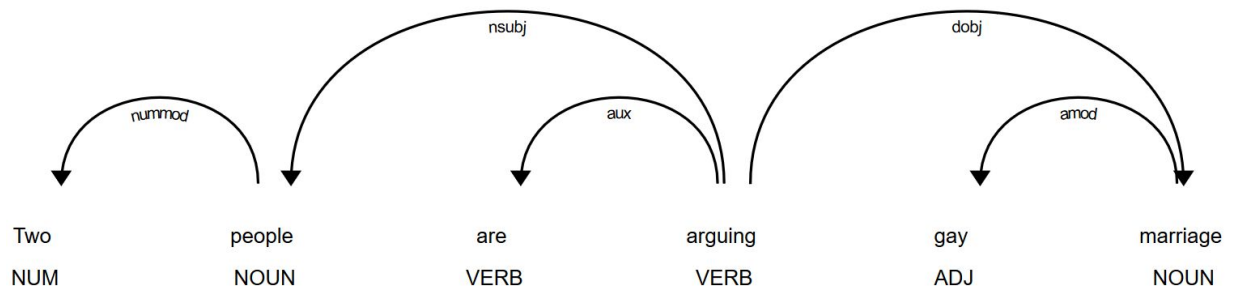


Figure 16: Sample dependency parse tree using spaCy library

In order to find a way to use these dependency trees to create the ERE triples we analyzed the trees formed for a lot of different sentences. We looked for patterns that we could use to generate entities and relations. The first thing we noticed is that, in the most basic of cases (like the sentence above), often the root of a sentence forms the relation. This relation may have important dependents that further the relation, things like negations.

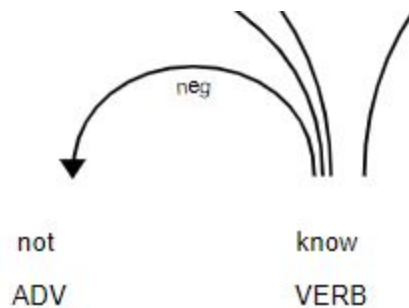


Figure 17: Sample sentence root with negation dependent

Our plan was to take this root text, along with a few special dependents like negations, and form the relation. We then needed to find the two connecting entities. We noticed two major patterns to find these entities. It seemed like the main thoughts that came off of the root (the thoughts that we wanted to put into our EREs) were a dependent subject and then a dependent complement. The subject was usually connected by a clausal subject or nominal subject dependency relation. The complement was usually connected by a complement dependency relation. So our initial idea was to take the subject connected to the root, traverse its subtree to find an optimal subtree, and use that as our first entity. We would then do the same for the subtree connected to the complement. We would then have our two entities and relation to form our ERE for a given sentence. For our example above, the sentence “two people are arguing gay marriage”, the root is the word “arguing”. We would take this as our relation. We would then look for a subject dependency relation and find the word “people”. We would take “people” along with its number modifier “two” to form our first entity. We would then look for our complement off of the root. In this case it takes the form of a direct object dependency relation with the word “marriage”. We would take the word “marriage” along with its modifier “gay” to form the second entity. Finally, we would create the ERE (“two people”, “arguing”, “gay marriage”).

This formed the basic idea for using the dependency parsing for generating EREs. However, due to the fact that these conversations were written on an internet forum, the pattern we found often did not fit the sentence. The first big problem we ran into was conjunctions. There were two cases of conjunctions that we needed to handle. First was the conjunctions that came off of the original root of a sentence. These conjunctions would often lead to their own ERE in the sentence that would be independent from the relation formed from the root. So that is how we processed it. If there was a conjunction that came off of the root of the sentence, we treated that as its own root and processed that subtree separately. The second type of conjunction was a conjunction within the complement. These conjunctions would often lead to EREs that shared the subject that came off of the root, along with the relation from the root. So to process these we would go to the complement and process the subtree. Any conjunctions would be treated as new entities attached to the original subject entity and relation. However, these two types of conjunctions would often get nested with each other, due to the fact that the conversation sentences often were run on sentences. It became quite unpredictable what would come of a sentence with many conjunction dependency relations.

Conjunctions were not the only problem within the dependency parser trees, however. Due to the fact that these conversation sentences came from an online form, the structure of a sentence was often unpredictable. There would be cases of seemingly random dependency relations being formed where there definitely should not be one. The spaCy library was just not trained to work with casual and unstructured text like ours often was. When properly using dependency parsing, many different patterns would need to be created, along with ways to recognize these patterns. A sentence could then be analyzed and the proper pattern could be used. There is not just one pattern that will fit any of these sentences.

Identifying referenced location

The ultimate goal of the project is to use knowledge graphs to identify what part of a conversation a summary is referring too. In order to do these we move to the final stage of our process. In it we use techniques such as cosine similarity to test our model and ultimately evaluate its effectiveness. Details on how this was implemented along with overall results can be found in the next section.

Testing/Evaluation Assessment

Testing and evaluation is done using the previously discussed sentence to dialog values. These values are passed into a lookup table for each summary. Each sentence can pair to a single, multiple or no pieces of dialog. In the case of pairing to no pieces of dialog that summary sentence will not be evaluated. In other cases each guess can be checked in order to verify that it pairs to one of the correct dialogs. If it pairs correctly to a piece of dialog we consider that to be a hit. If it ties multiple values, some of which are correct, we consider that a partial hit.

These hits and partial are then totaled for each of the 1294 evaluated sentences. A score is given considering the sum of hits divided by the total number of summary sentences. As shown below our code performs with 33.18% correctness while our random and spaCy baselines perform at 16.96% and 24.87%, respectively.

	Random Baseline	spaCy Baseline	Our Method
% Correct	16.96%	24.87%	33.18%
% Relative change over random	0%	46.63%	95.63%

Figure 18: Results of summary sentences to dialog matching

Defining Random Baseline

We defined our ultimate baseline as simply guessing the dialog for every single sentence. For each sentence it received a constant amount of points based on the number of correct dialogs divided by the total number of dialogs for each sentence. For this reason while we are defining it as random it does not actually randomly guess but instead returns the statistical median/mean value that would most likely be achieved by guessing. This provided an absolute baseline to compare with and check if our methods were at all successful.

spaCy Baseline

Our second baseline was created using spaCy. This was evaluated in the same way that our actual method was, where it generated a guess for each sentence. However, instead of using knowledge graphs or any of our analysis, it simply compared whole summary sentences to whole dialogs. It did this using spaCy's similarity function and then selected what sentence it considered to be the most similar as it's guess. This achieved notably better results than random chance, showing that NLP techniques are usable on our dataset for this problem.

Our Method

Our method was a combination of 3 separate methods. Firstly, TFIDF which stands for term frequency inverse document frequency was used on the summary ERE's in order to compare them to the dialog. TFIDF is a way of weighting vectors in order to compute a cosine similarity. In our case, each word within the ERE and within the dialog can be converted into a matrix which represents the total number of all word occurrences in the ERE/dialog. These word count vectors can then be scaled by the frequency of each word across the entire conversation. This way unique words like "Anglo", "Marriage", and "Israel" will be weighted more heavily than words like "the", "it", or "you". This is run for each summary ERE from the dialog and a similarity score of between 0 and 1 is given. For a more detailed description of TFIDF and the implementation we used, refer to [7].

Secondly we parse all dialogs for words that are considered rare. In this case we defined words that appear 3 times or less across the entire corpus to be rare. Each ERE is parsed for these words if they also appear in a dialog. Each ERE dialog pair is then incremented for each rare word pair found within each of them. Afterwards each value is divided by the maximum number of rare words in the ERE dialog pairs. This also gives a value from 0 to 1 for each ERE and dialog.

These two methods are run on every summary ERE. Each method is given a general weight (currently .8 for TFIDF and .2 for rare words) and the scores are totaled. The dialog considered most similar depending on those values is then given a higher value to be guessed. This is run for every ERE that comes from a single sentence. These values are then passed to the third method which uses them to make a dialog guess for the entire summary sentence instead of each ERE.

The last method is the label count and order method. We found that there were many dialogs where the summarizer directly referred back to S1 and S2. We found that when summarizers referred to S1/S2 multiple times within a given sentence they were much more likely to be referring to a dialog by S1/S2. In addition when summarizers mentioned both S1 and S2 they more often were referring to a dialog by the first person mentioned. Because of this we give each dialog by S1/S2 additional weight if the summarizer mentions them first and/or more times than the other person.

```
S1 acknowledges that he considers such religious objections to be evil.
S1 feels S2 and others like him/her support society's oppression of certain groups.
```

Figure 19: Examples of summarizers using S1 tag

Once all these values are computed we make a guess based on the dialog with the highest score. In the case of dialogs being tied, we check all dialogs and assign a number of points based on how many of the tied dialogs are correct. For example if two dialogs are tied but are both correct then full points are still awarded. If two dialogs are tied and only one is correct, then half a point is awarded.

Overall Results

As previously shown our method was able to outperform randomness and outperform an established method in the form of spaCy similarity calculations. While 33% accuracy is lower than we would have hoped, we believe that these results show that our methodology has merit and with additional experience and time could outperform established methods on additional datasets and examples. We believe that going into the future, these methods should be experimented with in more detail and with larger projects. Being able to create an ERE focused and forum post focused NLP library similar to spaCy may improve the results of this method greatly.

User's Manual

This section aims to help anyone set up the project and access the needed files.

Dependencies and Project Setup

Install *Python 3.7* (<https://www.python.org/downloads/release/python-373/>) and then follow the following dependencies instructions.

Python Package Installation

- (1) Install NLTK and needed NLTK language packages
\$ pip install nltk
- (2) Install spaCy and needed spaCy language packages
\$ pip install spaCy
\$ Python -m spaCy download en
\$ Python -m spaCy download en_core_web_md
- (3) Install Python requirements
\$ pip install -r requirements.txt

GraphViz Installation

GraphViz is the tool used to generate the graph png files. To use it with Python 3 you must add the executable to your path environment variable.

- (1) Download the GraphViz installer from https://graphviz.gitlab.io/_pages/Download/Download_windows.html
- (2) Run the installer. The default installation directory is 'C:\Program Files (x86)\GraphvizX.XX\bin'
- (3) Open the Windows Control Panel, navigate to *Control Panel -> System and Security -> System*, on the left hand side of the window click on *Advances System Settings*.
- (4) Once there, click on *Environment Variables* towards the bottom of the window.
- (5) Select the *Path* variable for your user or for the system, depending on which you are using for the Python 3, and press *Edit*.
- (6) On the right hand of the new window press *New*, and enter the path for the GraphViz bin, 'C:\Program Files (x86)\GraphvizX.XX\bin'.
- (7) Run the *graphviztest.py*. An outcome message will be printed.

Github

Clone the git repository from the Github [9] link. All of the following instructions can then be executed within the project folder.

Running the Project

Front End Product

The best way to view the project results is through the front end localhost HTML page described above. To run this file navigate to the project folder in a command terminal and type:

```
$ python runflask.py
```

You can then navigate to *localhost:5000/* in any web browser to view the results. This page is meant to be used as a tool to view all of the results of the project. Previously we had to dig through the data file, png files, and everything else when looking for results. This page has it all in one place. The knowledge graphs and other data are generated in real time, so any changes made to code can be seen in this page with a refresh. The following screenshots go into the functionality of this page.

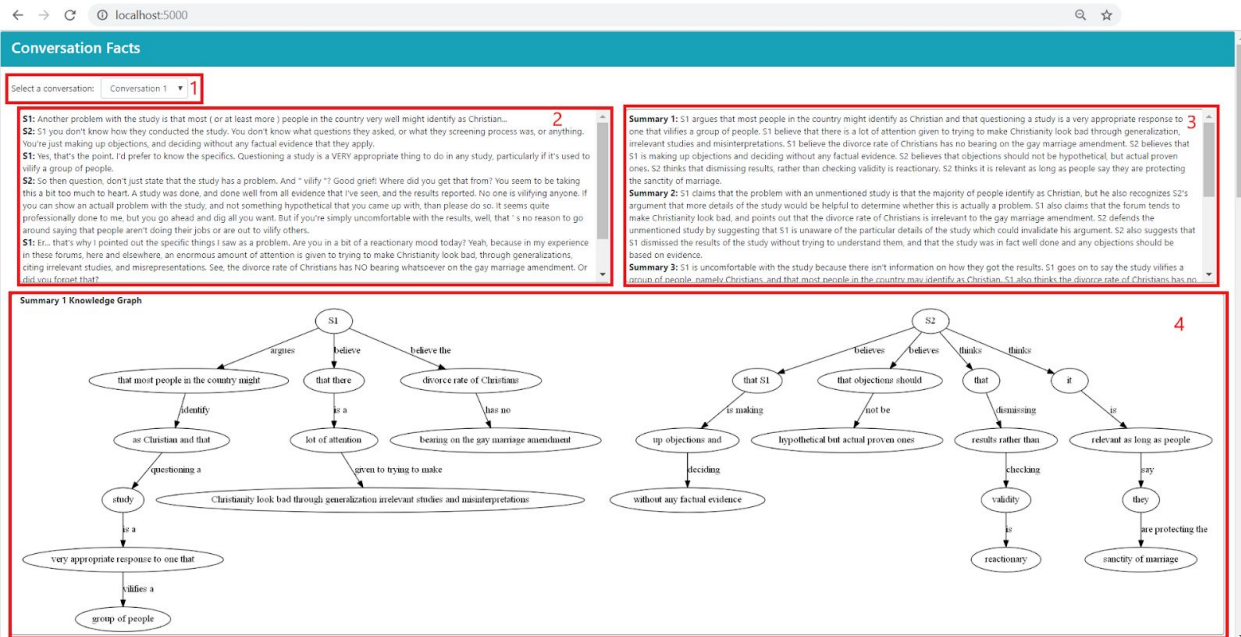


Figure 20: Overview of the front end html page

The page is broken into four main pieces that will each be explored further. First is the conversation selection, labeled 1.

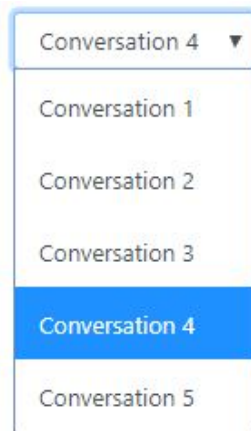


Figure 21: Front end html conversation selection (1)

Every conversation in the dataset can be selected. After a conversation is selected the page is updated and populated with new data, including the conversation text, summary texts, and generated knowledge graphs.

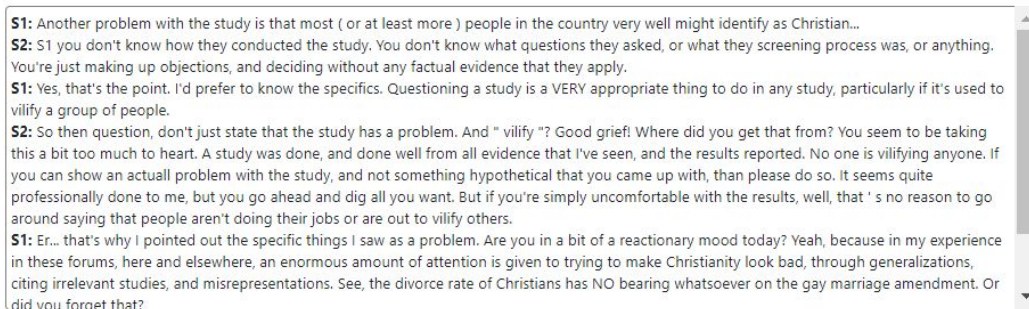


Figure 22: Front end html conversation text (2)

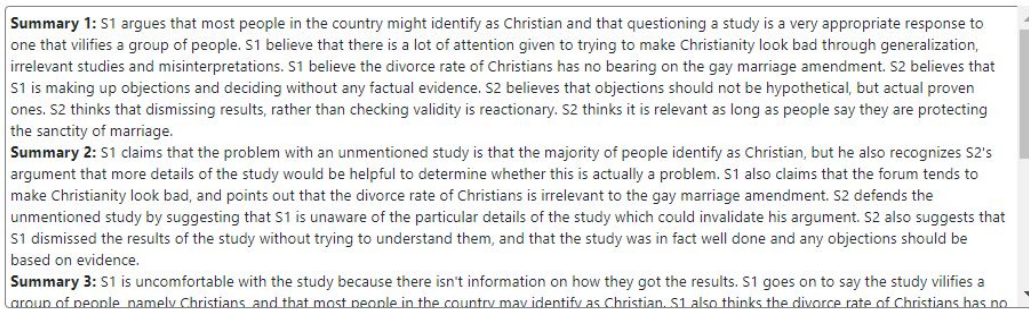


Figure 23: Front end html summary texts (3)

The selected conversation is loaded and placed right next to the five summaries of that conversation. This allows for the user to easily match sentences between the two in your head. Below this then follows the five knowledge graph images for each summary.

Running Specific Files

Many files within the project can be run on their own. Below is a table of the runnable files with their arguments, along with what to expect.

Table 1: List of executable files

File Name	Arguments	Expected Output
depparser.py	<id> - id of conversation	The conversation text along with created EREs
summ_depparser.py	<id> - id of conversation	The summary text along with created EREs
graphviztest.py	none	A message confirming graphviz is installed
runflask.py	none	Localhost at port 5000 with front end
parse.py	<id> - id of conversation	Formatted conversation and summary text
generatekgs.py	<id> - id of conversation	.png files for the conversation and summary knowledge graphs
evaluate.py	<id1, id2> - range of conversations to be evaluated	Number of sentences correctly paired for each summary and overall number of sentences correctly paired for all conversations in range
coreference.py	none	Prints example summary before and after coreference resolution

Docker

Docker is a free tool that allows users to create, deploy, and run applications through the use of containers. We have created a container image that contains all the dependencies and files of our project so that anybody can pull the image without having to go and download each library independently. To do this, you must first install Docker[] for your system. Once downloaded and installed, you should be able to run the command:

```
docker run -it -p 5000:5000 cjk6/cs4624:conversationfacts /bin/bash
```

This should pull the image from the repository and start a /bin/bash prompt. Then, you should navigate to the VT_CS4624 file using:

```
cd VT_CS4624
```

From here, you may run any of the Python files with the specified arguments above with:

```
python <filename>.py [arg]
```

```
root@34fed04eecdd:/app/VT_CS4624# ls
README.md      data.json      evaluate.py    parse.p        runflask.py
__pycache__   data.raw       generatekgs.py parse.py        summ_depparser.py
coreference.py depparser.py  graphviztest.py requirements.txt templates
```

Figure 24: Docker container

If you want to see the application after running ‘python runflask.py’ then type *192.168.99.100:5000* into a web browser.

Developer's Manual

This section aims to help future developers on the project. The project was created on Windows machines, all installation instructions are meant for Windows. Clone the git repository from GitHub [9].

File Inventory

The following table contains all files in the project along with a short description.

Table 2: List of project files

File Name	Description
data.json	json file of the argumentative dataset referenced before
data.raw	json file of raw original data
coreference.py	original work put into fixing coreference problem
parse.py	parses and cleans json data from data.json to be used in other files
parse.p	Cached pickle object file created when parse.py is run
runflask.py	runs a localhost html page to view data and results
chunk.py	generates the knowledge graph for a specific conversation summary
generatekgs.py	generates the knowledge graphs and EREs for each conversation
analyze.py	generates the parse tree used in the initial implementation
evaluate.py	matches the EREs for conversations and summaries and evaluates
depparser.py	generates the EREs for conversations using the dependency tree
summ_depparser.py	generates the EREs for summaries using the dependency tree
graphviztest.py	verifies that graphviz is installed and working properly
requirements.txt	holds the Python requirements for the project
readme.md	readme for project
graphs folder	holds the knowledge graph '.png' and '.dot' files
templates folder	holds the '.html' files used in the flask server
index.html	front end html page used to display all of the data

Dependencies and Project Setup

Refer to the previous section for installation requirements and instructions.

Project Overview

Front End Product

The front end HTML page runs with Flask. Flask is a microframework for Python based projects that allows for easy setup of a localhost server. Our frontend project is made up by one html index page and four Flask routes that handle the different requests. The four routes are:

- | | |
|---|--|
| (1) <code>'/'</code> | This renders the <i>index.html</i> page |
| (2) <code>'/conversation/<ind:id>'</code> | This returns the conversation JSON data for the specified id |
| (3) <code>'/summary/<ind:id>'</code> | This returns the summary JSON data for the specified id |
| (4) <code>'/kgs/<ind:id>'</code> | This generates the knowledge graphs for the specified id |

The HTML page is formatted using Bootstrap and the data is gathered using JQuery AJAX calls. When a new conversation is selected, each route is called (except the first one) and then the returning data is processed and passed out to the appropriate HTML divs. For a further look into how the HTML page is generated look at the *index.html* file located in the *templates* folder.

Pipeline

The overall project works in a pipeline manner. Many of the files build off of each other with the final executable being *evaluate.py*. The pipeline starts with *parse.py*. This file takes *data.json* and turns it into a list of dictionaries with all of the parsed data. Figure 25 is an example of part of the *data.json* file.

```
{
  "key": "1-5670_24_21_26_29_30_32_2",
  "Dialog": "S1:1- Another problem with the study is that most ( or at least more ) people
  "Summary": "-----\n D0\n -----\n S1 argues that most people in the country might
  "Summary_to_dialog": [
    [
      "0,0",
      "0,2",
      "1,4",
      "2,4",
      "3,1",
      "4,4",
      "5,5",
      "6,5"
    ]
  ],
}
```

Figure 25: Example dictionary in *data.json*

The *key* attribute is the conversation key generated by the original creators of the dataset. The *Dialog* attribute is the conversation dialog, to see how this is parsed look to the parsing section of this paper. The *Summary* attribute is the summary text: this makes up five summaries of the conversation. To see how these summaries are parsed, look to *parse.py*. Finally, the *Summary_to_dialog* attribute contains the manually annotated data.

Parse.py takes this data.json object and creates a ‘Entry’ object from it.

```
class Entry:
    def __init__(self, key, dialog, summary, summary_to_dialog):
        self.key = key
        self.dialog = dialog
        self.summaries = summary
        self.dialog_raw = dialog
        self.summary_to_dialog = [dict(), dict(), dict(), dict(), dict()]
```

Figure 26: Entry object created in *parse.py*

This entry object has most of the same properties as the original JSON data with the addition of the raw dialog for the Flask front end product. To get a better idea of how *parse.py* actually parses the data look to the commented code and the section in this paper. *Parse.py* is then imported into the rest of the files in the pipeline and the data is received using ‘*parse.parse()*’.

After parsing the data, the EREs are generated. *Generatekgs.py* creates the EREs and knowledge graph PNGs for each conversation. This file has two functions ‘*generate_kgs*’ and ‘*generate_kgs_depparser*’ both of which take the desired conversation ID. ‘*generate_kgs*’ uses chinking and chunking to generate the results, while ‘*generate_kgs_depparser*’ uses the dependency parser files to generate the results. Both of these functions open a .dot file and text file for each summary and the conversation. Each ERE is written to the text file along with what sentence/dialog it comes from. The .dot file is used to generate the knowledge graph PNGs. Graphviz is a technology that takes these .dot files and generates PNGs. For further information on how Graphviz works, look to their website (<https://www.graphviz.org/>). Simply put, the entities of the ERE are created as nodes, and the relations of the ERE form the edges. This then creates a spanning graph. The generated text file is used later in the *evaluate.py* file.

Generating the EREs can be done in one of two ways, chinking/chunking or dependency parsing. For how both of these functions are implemented look to the respective implementation sections. The actual code for both of these will now be covered. The chinking/chunking method is found in *generatekgs.py* along with the rest of the code covered above. The dependency parsing is done in two different files. *Depparser.py* parses the conversation dialog using dependency parsing. *Summ_depparser.py* parses the summary text using dependency parsing. Both of these files work in a very similar way. They both generate the spaCy dependency parser, discussed in the implementation, and then recursively process the subtrees of the root word. There are many different relations that these subtrees can have and different dependency relations that are needed within a subtree. Depending on the type of dependency relation the entities are formed and joined by the different relations. The final EREs are matched and evaluated in *evaluate.py*. The process of how these matches are made is discussed in the implementation section.

Lessons

Verifying library abilities is important

One of the first problems we ran into was either not knowing the full capabilities of a given library or overestimating them. One major example of this was trying to use the SNAP.py library (<https://snap.stanford.edu/snappy/>). We wanted to be able to use a library to handle the drawing of all of our knowledge graphs and this seemed to offer a solution. After spending large amounts of time getting Python 2, SNAP.py and its dependencies, learning the software and editing our current implementation to work smoothly with both Python 3 and Python 2 libraries, we realized it was missing core functionality. Namely, with SNAP.py you are unable to label edges in a given graph. This is of course essential for our implementation in our representation of ERE triples.

Thankfully this was able to be resolved using Graphviz. Ultimately SNAP.py uses Graphviz to generate .png files so we were lucky enough to bump into it. We are able to algorithmically generate our own .dot files and then use Graphviz to generate PNGs.

Evaluation of NLP techniques is difficult

One of the hardest recurring themes within this project is the difficulty of evaluating our progress. You can look at knowledge graphs and say one is better than another but it is hard to evaluate them in mass. Every change you make may solve a problem you have identified in one graph but create many more problems in others. For example we initially removed stopwords and found that it greatly helped many graphs become more condensed. However it also removed words such as “not” which are absolutely essential when discussing a person's argument. In addition it is very difficult to evaluate a given knowledge graph from a summary without reading the entire summary. This often leads to over analyzing a select few summaries and ultimately lowering the quality of the entire pipeline by making poor choices. We didn't have a mathematical way to evaluate overall quality of knowledge graphs until the end of the project which ultimately slowed progress.

Not all NLP methods are useful for every problem

As stated above, naive stopword removal was initially considered but ultimately had disastrous results. Going into this project we considered many different techniques and wanted to

use most of them. However, while every NLP technique has a benefit they all each have a downside. Lemmatization and stopword removal will reduce context. Naive dependency parsing leads to choosing poor parts of speech as edges. Constituency parsing doesn't remove unneeded parts of a sentence. While we initially thought we could just use every technique and receive the benefit we didn't fully consider the cost of each technique. Moving forward we were more careful evaluating both the pros and cons of adding any new method to our pipeline.

Processing internet text conversations is a challenge

As previously brought up in different parts of this report, there are a lot of changes that came up with handling the conversation side of generating the EREs. These conversations come from internet forums where people type whatever they want with very little consideration of grammar and sentence structure. There was a noticeable difference in the consistency of sentence structure between the summaries and conversations. Obviously the summaries are written to be more proper, while the conversations are very casual. These casual dialogs often led to what seemed like random sentence structures which made finding patterns in the dependency trees very hard to do. With beginner level NLP skills and a limited amount of time, it was very difficult to create satisfying EREs from the conversations. That being said, there are benefits to using these EREs to match summary sentences and conversation dialogue, and perhaps having a better method of generating these EREs would be very valuable.

Off the shelf libraries don't work for all cases

Throughout the semester, we had tried to use several technologies to assist in creating the project. Some of them were useful which we kept in production, but some did not work for us. This caused setbacks during development as we had to try to get them to work but eventually decided that it they would not be needed. The first difficulty we encountered was trying to run a visualization tool called snap.py. This is a Python interface that is used to generate graphical networks which we would have used to help us represent our Knowledge Graphs. However, there were several problems installing this since it requires Python 2.7.x whereas our project ran on libraries that were dependent on Python 3.7. Because of this, we decided to use Graphviz [6] to create our graphs, as it did not require the same Python version.

Another set of tools that we looked into was Python libraries for coreference resolution. These libraries are machine learning models that are trained to resolve ambiguous entity references within text. Two such libraries are spaCy's NeuralCoref library and AllenNLP's coreference model. We had tried using both of these to resolve our coreferencing problem but neither of them was able to accurately resolve the entities. We believe that this is due to the

peculiarity of our dataset as there were many unusual sentence structures and incorrect syntax in the data. As mentioned previously, this required us to create our own coreferencing methods.

Future Work

This section aims to provide further ideas to anyone who is trying to tackle the same problem. Due to time constraints there were some things we could not try. Our final results were not great, but they were promising. We believe that the processes that we explored in this paper could be expanded on to get better results. Most of the issues we came across related to the actual generation of ERE triples. We believe that improving these would be the first step towards improving the results.

Dependency parsing was a good tool for generating EREs. However, we tried to find one pattern that matched all sentences in our dataset. This was a naive approach. We believe that dependency parsing could be more effective if more patterns are found. A set of patterns could be found, along with ways to detect which pattern a sentence fits into. From there you can just analyze each sentence depending on what pattern it fits into. This technique would obviously produce great EREs, however it does have its problems. You do not know how many types of patterns you would need. Especially when it comes to internet threads, text can be very unpredictable. Along with the problem of quantity, it is quite tedious to examine all of the text in a dataset and find the patterns. Dependency parsing has its benefits but it can be a tedious tool to use.

There were many other NLP tools that we looked into, but did not try too hard to implement. One of these in particular was constituency parsing. Constituency parsing aims to extract a constituency-based parse tree from a sentence that represents its syntactic structure according to a phrase structure grammar. We looked at the constituency trees for a few different sentences, however we did not see an easy way to use it to our advantage. However, we have no doubt that there are also patterns within the constituency trees. These patterns could be analyzed in order to find a way to create the ERE triples.

The final thought we had about future work had to do with the use of machine learning. Using any of the above techniques, a machine learning algorithm could be taught to further improve the generation of EREs. None of us had any experience with machine learning so nothing was attempted.

We think that there is great potential for using EREs to match summary sentences with dialog sentences, and we hope that our initial work can help any future attempts at this problem.

Acknowledgements

Client: Saurabh Chakravarty (Rackspace Software Developer, VT Ph.D Student)
saurabc@vt.edu

Advisor: Dr. Edward Fox (Virginia Tech Professor)
fox@vt.edu

References

[1] Main data sets: Argumentative Dialogue Summary Corpus: Version 1.
Amita Misra, Pranav Anand, Jean E. Fox Tree, Marilyn Walker. "[Using Summarization to Discover Argument Facets in Online Ideological Dialog](#)", in *The North American Chapter of the Association for Computational Linguistics (NAACL), Denver, Colorado, 2015*.

[2] Christopher Manning (Professor at Stanford University)
Online lectures on NLP concepts and techniques, 2019
https://www.youtube.com/watch?v=OOQ-W_63UgQ&list=PL3FW7Lu3i5Jsnh1rnUwq_TcylNr7EkRe6

[3] Evaluating open relation extraction over conversational texts
Imani, Mahsa 2014
<https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0165856>

[4] Learning Knowledge Graphs for Question Answering through Conversational Dialog
Published in HLT-NAACL 2015
<https://aclweb.org/anthology/N15-1086>

[5] spaCy API
<https://spacy.io/api>

[6] Graphviz Documentation
<https://www.graphviz.org/documentation/>

[7] Gensim TFIDF Model
<https://radimrehurek.com/gensim/models/tfidfmodel.html>

[8] NLTK
<https://www.nltk.org/>

[9] Github Repository
<https://github.com/jdm2980/ConversationFacts>

[10] Docker
<https://www.docker.com/>

Appendices

A.1 NLP Parts of Speech

Table 3: NLTK Parts of Speech

POS Tag	Part of Speech
CC	Coordinating conjunction
CD	Cardinal digit
DT	determiner
EX	Existential there
FW	Foreign word
IN	preposition/subordinating conjunction
JJ	Adjective ‘big’
JJR	Adjective, comparative ‘bigger’
JJS	Adjective, superlative ‘biggest’
LS	List marker
MD	Modal could, will
NN	Noun, singular ‘desk’
NNS	Noun plural ‘desks’
NNP	Proper noun, singular ‘Harrison’
NNPS	Proper noun, plural ‘Americans’
PDT	Predeterminer ‘all the kids’
POS	Possessive ending parent’s
PRP	Personal pronoun I, he, she
PRP\$	Possessive pronoun my, his, hers
RB	Adverb very, silently
RBR	Adverb, comparative better

RBS	Adverb, superlative best
RP	Particle give up
TO	To go ‘to’ the store
UH	interjection
VB	Verb, base form ‘take’
VBD	Verb, past tense ‘took’
VBG	Verb, gerund/present participle ‘taking’
VBN	Verb, past participle ‘taken’
VBP	Verb, present, non-3d ‘take’
VBZ	Verb, 3rd person ‘sing’, present ‘takes’
WDT	WH-determiner ‘which’
WP	WH-pronoun ‘who’, ‘what’
WP\$	Possessive WH-pronoun ‘whose’
WRB	WH-adverb ‘where’, ‘when’

A.2 spaCy Syntactic Dependency Labels

Table 4: spaCy dependency relations

POS Tag	Part of Speech
ACT	Clausal modifier of noun
ADVCL	Adverbial clause modifier
ADVMOD	Adverbial modifier
AMOD	Adjectival modifier
APPOS	Appositional modifier
AUX	Auxiliary
CASE	Case marking

CC	Coordinating conjunction
CCOMP	Clausal complement
CLF	Classifier
COMPOUND	Compound
CONJ	Conjunct
COP	Copula
CSUBJ	Clausal subject
DEP	Unspecified dependency
DET	Determiner
DISCOURSE	Discourse element
DISLOCATED	Dislocated elements
EXPL	Expletive
FIXED	Fixed multiword expression
FLAT	Flat multiword expression
GOESWITH	Goes with
IOBJ	Indirect object
LIST	List
Mark	Marker
NMOD	Nominal modifier
NSUBJ	Nominal subject
NUMMOD	Numeric modifier
OBJ	Object
OBL	Oblique nominal
ORPHAN	orphan
PARATAXIS	Parataxis

PUNCT	Punctuation
ROOT	Root
VOCATIVE	Vocative
XCOMP	Open clausal complement

A.3 Meeting notes

Table 5: Meetings notes and goals

Date	Topics of discussion	Goals until next meeting
2/1	<ul style="list-style-type: none"> Dataset - choose one dataset Possible approaches using NLP techniques 	<ul style="list-style-type: none"> Go through all the materials Finalize the dataset Come up with one idea as a team Next meeting on 2/11 at 7:30 PM at KW-II
2/11	<ul style="list-style-type: none"> Finalize dataset Clarify any concepts and check the understanding of the team Check the familiarity with the libraries (NLTK/spaCy) Review the idea by the team Send link to try out some NLP concepts before starting an implementation Generate KG out of a conversation Generate a nice looking KG Sentence Chucking 	<ul style="list-style-type: none"> Refine the updated plans Hand made KGs of at least 4 summaries Try out chunking and identify the various regex patterns to identify (e,r,e) tuples Start thinking about how to generate KGs from a conversation. Hints(stopwords removal, lemmatization, dependency parsing) Collaborate!!!
2/15	<ul style="list-style-type: none"> Example KGs for summary text KGs for conversation <ul style="list-style-type: none"> Much harder problem Cosine similarity for comparing strings Current approach, generate KGs for summary. Search for entities within the conversation 	<ul style="list-style-type: none"> Generate KGs of summary From KGs, find ERE tuples in the conversation by hand Send 5 sets of conversations and summaries from the dataset