Event Ordering and Analysis

Dalton Hubble and Thomas Garver

6.863 Final Project

December 14, 2012:

1. Introduction

In our final project, we sought to build a system to analyze written texts and to build a software representation of both the time and relative ordering of the different "events" described in the text. We chose to focus on simple events that described the actions of significant entities such as people or organizations. The bulk of our research and implementation efforts were dedicated to event time extraction and event ordering. Inherent in tackling this challenge was the problem of "event" extraction from a body of text - this in itself is a hard NLP problem related to text summarization. The relevant concepts, ideas we introduced, and approaches we took to event time extraction and inferring event orderings are described in detail in following sections, along with some of the simplifying assumptions we made about sentence structures and the events in them to make the event extraction problem tractable.

We implemented a system to process a body of text and build a queryable datastore of time and relative ordering information corresponding to the events identified in the body of text. A simple query interface and simple query language were defined for making queries to determine the time at which different events occurred or to provide the relative ordering of two given events. To perform time queries, users could describe an event found in the text and receive the best available information (drawn from information spread across the entire text) about the date/time at which the event occurred. Similarly, to perform order queries, users could describe two events from the body of text and receive the relative ordering of those two events based on both exact time information about each event and using textual clues throughout the text to relate events (such as 'until', 'after', 'later').

We believe a system like this illustrates a number of the challenges of this domain, while showing that systems can be built to "read" a text, extract time and ordering information from it, and provide that information via a query-able interface.

2. Concepts

2.1 Events

To begin to analyze the timing and ordering of events, a method to extract the essential events of a body of text was needed. This problem in itself is a significant unsolved natural language challenge as it requires a solution to the text summarization problem as a subcomponent.

First it is necessary to know which parts of a sentence correspond to different "events" or what a human reader would consider to be an event. In the sentence:

Fred shoveled snow until Alice told him to stop.

it is necessary to extract out the event of “Fred shoveled snow” and “Alice told him to stop”. There is no purely syntactic breakdown of a sentence which contains multiple events inside of it. Not all sentences contain the type of “events” that we are looking for, in which an entity such as a person or organization performs an action. Consider the following sentence which most people would consider not to contain an event (although in some contexts this can be an event)

Flowers are a decoration.

So there is some ambiguity about how to define an event in a sentence. Additionally, multiple events can interact and intertwine in ways that make them difficult to pull apart and extract. For example, the events of the sentence:

Fred, until Alice told him to stop, shoveled snow, but later drank some hot chocolate.

contains 3 events which are not in contiguous phrases. Existing parsers are not perfect and cannot consistently provide the ideal parse tree for separating out events. Moreover, even if a parse tree which is suitable for drawing event distinctions could be constructed it is unclear how to use syntactic clues in the parse tree to pull apart the different events in a sentence in a generally applicable way.

Further complicating the problem, it is necessary to know which parts of a phrase describing the event actually correspond to the real event. Many words in a sentence are “fluff” words which are not actually essential for expressing the real event that transpired. For example, consider:

Entering the United States Military Academy in June 1911, Eisenhower had a "spectacular" 1912 football touchdown praised by the New York Herald .

A human reader might summarize the “event” of this sentence as “Eisenhower had a great touchdown”. Clearly not all the words in the sentence are necessary to represent the event itself.

Finally, there is also the challenge of only identifying events in which some meaningful action has transpired. In a great deal of prose text, many sentences are devoted to describing objects, but do not contain actual actions. The previously sentence about flowers is an example of a sentence in which no action is performed. Humans consider events to be actions related to specific people, organizations, or object so it can be difficult to distinguish between sentences that are statements and sentences describing events.

Rather than solve this challenge head on, which would have been a project in its own right, we made some simplifying assumptions about the events we wanted to consider, developed a naive event extractor, and focused on extracting time and relative ordering information. All considered sentences were required to contain either one or two events. We detected sentences with two events if the sentence was contained a subordinating conjunction or some other part of speech to create a sentence phrase S or SBAR nested inside the sentence parse tree. This nested phrase was considered one event while the remainder of the tree was considered the other event. This worked well for the simple examples we wanted to consider such as

Fred shoveled snow until Alice told him to stop.

If no such nested sentence phrase was found in the sentence, the sentence was assumed to be a simple sentence containing a single event. Entity tagging was performed on all sentences to identify significant entities.

We did not seek to build a system to process any general sentence and instead focused on analyzing simple sentences with one or two events and entities performing actions. This de-stressed the event extraction components of the system and stressed our ability to perform event time and ordering extraction. If a superior event extraction module became available, our system would be ready to incorporate it to handle more advanced sentences and the extracted events. The core logic of event time extraction and event ordering would remain unchanged.

2.2 Reference Events

The whole idea of analyzing a passage of text is that the passage provides additional information about the relative orderings of events and about the timing of events over the course of reading. For example, early in the text we may learn that "In June, *eventA*, until *eventB*". This tells us information about the time at which *eventA* happened and tells us that eventA happened before *eventB*. Much later in the text, additional clarifying information might be provided. For example, *eventA* ocurred on Tuesday at 9:00AM and occurred after *eventC*. The additional timing information should augment the information already known about *eventA* and a new relationship in which *eventC* is followed by *eventB* should be established.

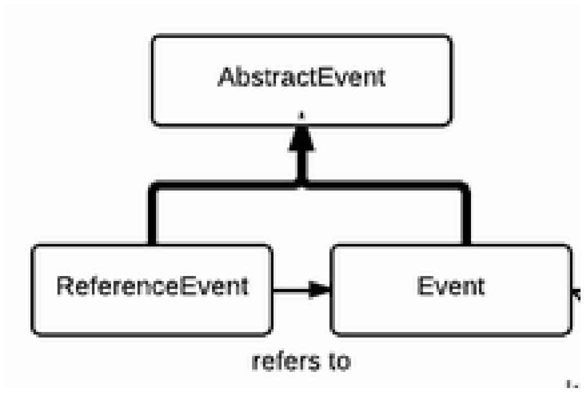


Figure 1: ReferenceEvents vs. Events

Clearly, then, not every (Abstract) Event in the body of text can be independent and there needs to be a way to pair up events that refer to the same physical phenomenon. To do this we introduced the idea of a ReferenceEvent. ReferenceEvents are pointers to Event objects which store all the information about the physical event (Fig. 1). In our system, we devised a strategy so that whenever we extracted an event[1] from a sentence, we would compare it to the list of already seen events and score how close the match was based on named-entity similarities and fuzzy string matching. If the extracted (Abstract) Event was similar to a previous event, it was assumed to be a ReferenceEvent. Otherwise, it became a new Event of its own. For example,

> *(Until the blizzard killed Bill on Friday), Alice was visiting the galleria.*
> *Fred shoveled snow after (the blizzard killed Bill.)*

In these two sentences, the parenthesized portions refer to the same event. Therefore, when our online text analyzer reaches the second sentence, it recognizes "the blizzard killed Bill" to be a reference to the blizzard that killed Bill Friday occurring in the first sentence. This reference allows the relative ordering.

Event(Alice was …) -> Event(blizzard killed...) -> Event(Fred shoveled...)

to be learned by the analyzer. It also allows the timing information about the blizzard occurring on Friday to be paired with the single Event(blizzard killed...). Without ReferenceEvents, chains of relative ordering would not be possible and this example would have two blizzards, one that occurred on Friday and another which occurred at an unknown datetime. Figuring out whether abstract events encountered in text are new events or references to previously seen events is essential for both timing and ordering.

2.3 Online Text Analysis

Now that we've discussed the Events and Reference events inside a text, let consider the general approach to extracting these AbstractEvents and extracting ordering and timing information. The design we proposed was to online textual analyzer that would process one sentence at a time (i.e. online) and extract event(s) from that sentence. If the sentence contained two events, the order in which the two events occurred would be preserved. From there steps could be taken to obtain ordering information and timing information learned from analyzing the current sentence. Extracted information would be used to augment and update two software representations of extracted information, a datastore for timing information and a datastore for ordering information. Then, the text analyzer could move on to analyze the next sentence. Once the entire text has been analyzed, the datastores would contain all of the merged and updated information learned about all of the events in the entire text. Queries could be performed on these datastores. See Fig. 2.

[1] Lowercase "event" is used in the general sense (i.e. to refer to an AbstractEvent) which may be an Event or ReferenceEvent. We'll specify Event or ReferenceEvent explicitly when needed.

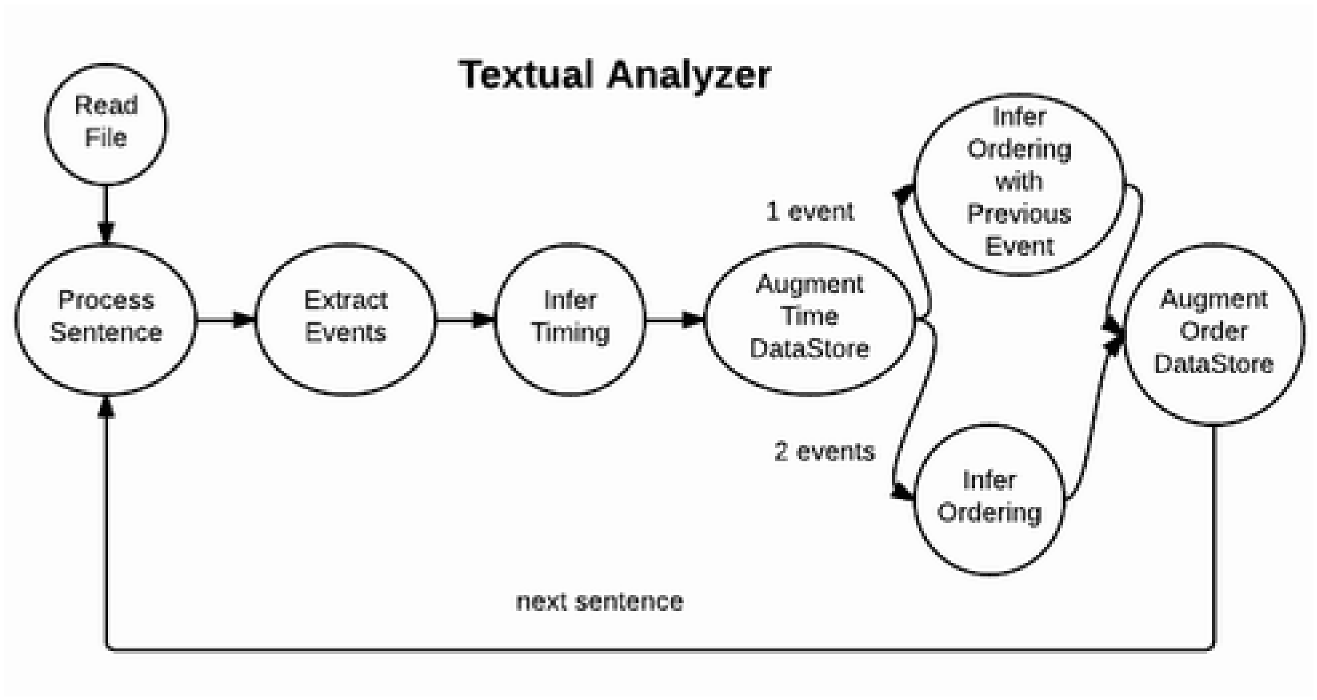


Figure 2: Textual Analyzer Flow Diagram

2.3 Ordering

2.3.1 Ambiguity for Sentences with Many(>2) Events

Considering the current sentence being analyzed by the text analyzer. Because of our requirements on the allowed sentences, the sentence will have either one or two (Abstract) events only. This requirement was largely put in place due to ambiguity that is introduced when a sentence contains 3 or more events in it. Consider for example:

*John ate a burger **after** Sam hugged the kitten **before** the kitten purred.*

The goal of our ordering analysis is to use word clues such as “after”, “before”, etc. to determine the order of the events, but here its ambiguous whether John ate the burger before the kiten purred or whether Sam hugged the kitten before the kitten purred. There is a similar ambiguity at the “after”. Allowing only simple sentences with fewer than three events removes this source of ambiguity.

2.3.2 Conjunction Word Ordering

Making the restriction to fewer than three events per sentence, a typical sentence we might now encounter is

*John ate a burger **after** Sam hugged the kitten*

The event extractor breaks this into Event(John ate a burger) and Event(after Sam hugged the kitten)[2]. The leading several words of the second event are taken such as “after Sam”. All words in this region are compared against two catalogs of conjunction ordering words. One category of words implies a chronological ordering (first event before later event). The other category of words implies an anti-chronological ordering. Examples of catalogs might be:

```
# A before B
CHRON_CONJS = ["before", "until", "till", "til", "so that"]
# A after B
ACHRON_CONJS = ["after", "now that", "as soon as", "since"]
```

The words in the beginning of the second event (using our event extractor, conjunction words end up on the second event) are compared with each word in the chronological and anti-chronological catalog to determine whether the two events are ordered in one of the two ways. If the scores of each of the catalogs are similar (ex. none of the conjunction clue words in the catalogs are found) then no ordering is assumed or added to the datastore. In this case, since the word “after” in “after Sam” is found in the anti-chronological category, the events are given an ordering like:

Event(after Sam hugged the kitten) -> Event(John ate a burger)

2.3.3 Leading Word Ordering

Although conjunction clue words are useful, we must also consider leading clue words used at the beginning of a sentence. Consider this sentence for example:

Until the blizzard killed Bill, Bill was playing outside.

Two separate events Event(Until the blizzard killed Bill) and Event(Bill was playing outside) are extracted from this sentence and the ordering is determined by looking at the first few words of the sentence “Until the”. These words are scored against a catalog of leading words that imply chronological orderings of the two events and a catalog of leading words that imply an anti-chronological ordering of the two events

```
# A before B
CHRON_LEAD = ['after', 'following']
# B before A
ACHRON_LEAD = ['before', 'until', 'till', 'til']
```

[2] Note that the inclusion of “after” in the Event is not problematic. When a user performing a query wishes to specify an event, he/she only needs to provide a string that roughly matches the text of the event. Getting the entity of the event correct in the description is much more important than including exactly every word in the Event.

Once again, if no catalog is the clear winner, no ordering is assumed because we want the textual analyzer to only represent information gleaned from the text, not by making risky guesses about ordering. In this event, the anti-chronological leading words catalog will score higher and the events will be ordered:

Event(Bill was playing outside) -> Event(Until the blizzard killed Bill)[3]

2.3.4 Two Event Sentences

We've discussed two strategies of figuring out an ordering among the two events in a sentence containing two events by using conjunction catalogs and leading word catalogs. Since a two event sentence could be written to use either style, the actual ordering is done by comparing any conjunction words against the conjunction catalogs and any leading words against the leading word catalogs. The catalogs are scored as before and the best scoring catalog is used to determine whether the events occur chronologically or anti-chronologically. If there are conflicts between the leading and conjunction catalog winners or if scores are too close, no ordering is assumed.

2.3.5 Ordering With ReferenceEvents

Up until now, we've discussed ordering of events (i.e AbstractEvents). In reality, we only draw relationships between Events since they maintain time and ordering information attached to them. If a ReferenceEvent is part of a sentence, the ordering analysis is done in the same way, but once an ordering is determined, the order relationship is created between the Events referred to by any ReferenceEvent. Recall the example,

(Until the blizzard killed Bill on Friday), Alice was visiting the galleria.
Fred shoveled snow after (the blizzard killed Bill.)

The first sentence is analyzed, two Events are created and the leading word "Until" is used to discover that the ordering relationship is anti-chronological. Thus

Event(Alice was...) -> Event(Until the blizzard...)

When the second sentence is considered, the AbstractEvents are determined to be anti-chronologically ordered. However, the second event is ReferenceEvent(the blizzard killed Bill) which refers to Event(Until the blizzard killed Bill on Friday). Thus the anti-chronological relationship is built between Event(Fred shoveled...) and Event(Until the blizzard...).
Event(Until the blizzard...) -> Event(Fred shoveled...)

[3] Once again, the Until inside the event is not significant. Querying whether Bill playing outside or Blizzard killed Bill will return the correct ordering. Users do not need to include the word "Until" even though it is technically inside the Event(Until the blizzard killed Bill).

Note that our use of reference events allows chains of relative orderings of events to be created so only allowing sentences containing fewer than three events does not limit the nature of the orderings of events that can be constructed. This is analogous to the Chomsky Normal form of CFG still being fully expressive of the CFGs constructed using rules with more than 2 RHS terms.

2.3.6 One Event Sentences

Ordering is still an important consideration in with events in single event sentences. Consider the 2 sentences

The Americans counterattacked with cannons on April 27.
Afterward, Bostonians were freed.

When two adjacent one event sentences occur, it is possible that the leading word clues imply an ordering that we can pick up on. Note that if the previous sentence contains two events then it would not be clear which of the two events Bostonians were freed after. Thus in cases where two single event sentences occur simultaneously, an attempt is made to figure out the ordering relationship.

In this example, when we analyze the second sentence, which contains *Event(Afterward, Bostonians were freed),* we can find a “across sentence” ordering with the event in the previous sentence (which also had a single event). This is done by considering the leading words of the one event sentence against the catalogs of chronological and anti-chronological one event sentence leading words.

```
# Chronological
CROSS_CHRON_LEAD = ["later", "afterward", "next", "then", "later"]
# Anti-chronological
CROSS_ACHRON_LEAD = ["previously", "beforehand", "earlier"]
```

Scoring to determine an ordering, if any is done similarly to before and ReferenceEvents are converted to the Events that they refer to when creating the ordering. In this example, the “afterward” creates a chronological ordering so:

Event(The Americans counterattacked...) -> Event(Afterward, Bostonians were freed)

2.3.7 Ambiguity of “Same Time”

During our design phase, we recognized that conjunction words like ["as", "during", "as long as", "when", "whenever", and "while" in two event sentences implied that the two events in the sentence occurred simultaneously. We chose to totally ignore any information about events happening at the same time because it leads to complex ambiguity problems. The heart of the issues is that at a high level view, every conceivable event happened at the same time. At a closer view, we discover that *eventA* and *eventB*, while they were in the same millenia, occurred in different years. While *eventC* and *eventD* occurred in the same year, later we learn that they occurred in different months. The concept of events happening at the "same time" is a complex on and representing this information and modifying it over time (which can lead to infinitely deep nesting) creates a complex hierarchical graph in which there are many different levels of "at the same time". We explored this briefly, but the domain was far too complex for a final project and would work as a better thesis or longer term research project.

2.4 Timing

In many types of text, especially historical texts, there is a wealth of knowledge about the the dates and times that certain events occurred. While it may be difficult to search through this text and determine when something happened, a process of extracting this information from the text and associating it with its corresponding event greatly simplifies this process. With this information present for an event, the user can simply query the system about when it happened and the system will be able to quickly find the associated date and time, if one is present. Take the following sentence for example:

John began his freshman year of college on September 6, 2005.

It's clear from this sentence that a person could search through the text containing the sentence and determine on what day John started college. However, to ease this process, our system recognizes the date *September 6, 2005* and stores the information as a Date object associated with the event. Now, the user is able to query the system about when John began college, and the system will return the associated date object. Storing the time that an event occurred is also helpful when comparing the relative ordering of two events.

As a feature of our system, we added the ability to recognize dates and times in the text that may not have all of the features needed to determine an exact point in time. For example, a date of *February 2003* may be present in the text, and while it doesn't represent an exact time, it does provide information about when something happened. However, this can lead to issues of ambiguity when comparing two dates that have different features associated with them. For example, consider two date entities of *February 3, 2012* and *February 2012*. If we try to compare these two dates to see which one occurred first, we run into a problem: it's not actually clear which one happened first, and one may actually occur *during* the other. Details on how we resolved these types of ambiguities can be found in the Implementation section.

3. Implementation

3.1 Summary

The system we implemented consists of a textual analyzer which takes a filename to read a text from and processed each sentence, extracting events, ordering events, and determining times for events. The analyzer maintains an OrderDataStore and a TimeDataStore which have methods that allow them to be queried. The software is invoked by running **temporal_analyzer.py** with a number of command line options to be described in Usage.

Once an analyzer instance has been built from a file or from a shelved file (a way of storing previously build analyzers persistently), users can enter interactive mode where they can manually call methods on the analyzer to access its OrderDataStore and TimeDataStore to make query calls. Alternately, when invoking temporal_analyzer.py, a query file can be specified which contains a sequence of queries to be performed. We created a mini-query language which allows users to construct ORDER_QUERY and TIME_QUERY types of queries.

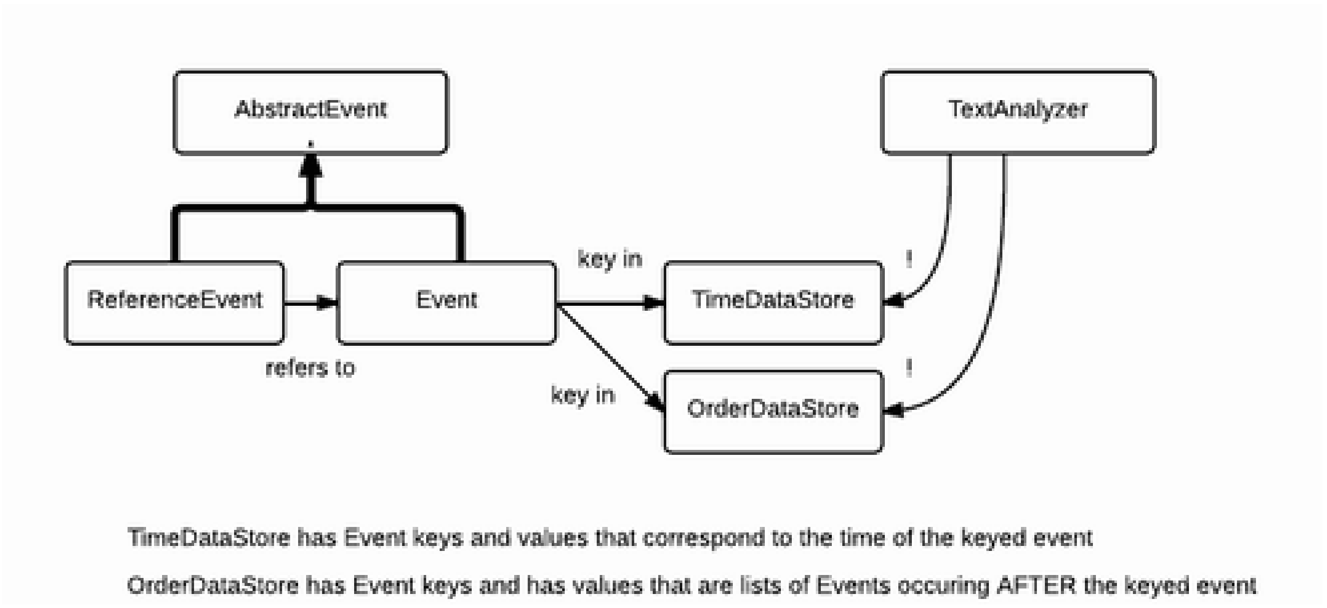

AbstractEvent
TextAnalyzer
ReferenceEvent
Event
key in
TimeDataStore
!
refers to
key in
OrderDataStore
!
TimeDataStore has Event keys and values that correspond to the time of the keyed event
OrderDataStore has Event keys and has values that are lists of Events occuring AFTER the keyed event

Figure 3

3.2 Parsing and Tagging

To perform many of the functions that our system is capable of, it is necessary to parse each sentence, tag the words of the sentence with their parts of speech, and extract named entities from the sentence's text.

Parsing is important because the sentence is broken up into its multiple events based on its parse tree. We found parsing to be a very tough challenge because of the lack of a good Python library that was able to parse the arbitrary kinds of sentences we were trying to parse. However, we are often very satisfied with the online Stanford parser (http://nlp.stanford.edu:8080/parser/index.jsp), though it does not have an associated Python package that we could install to perform our parses. Instead, we had to resort to a bit of "hacking" to allow us to use the great functionality of this parser from a normal Python script. We wrote a function to send an HTTP request to the parser website and then scraped the returned HTML to find the parse tree for our sentence. While this decision requires the system to have Internet access, the greatly improved performance of the parser means this is a tradeoff we were willing to make. **nltk.tree.ParentedTree** has a constructor that takes a parenthesized expression of the form (S (NP (...))) and constructs a tree representation of the parse automatically, which was very nice because this is exactly the format that the Stanford parser produces.

Part of speech tagging is not completely essential to the performance of our system, but the Stanford parser provided the POS tags automatically so they are included in our representation of the text. Named entity tagging is needed because entities are very important for our function that matches text with the closest Event. When trying to match a piece of text with an event, the function will only count a match if some of the entities in the Event and the text are the same. In this way, a piece of text like “Alice went to the store” will not match with the Event “Sam went to the store” because the entities do not match at all. Our system performs named entity tagging with the help of **nltk.ne_chunk**. This function returns a tree representation of the sentence, and from that tree representation we are able to extract entities to be stored with each Event.

3.3 Extracting Events from Sentences

As discussed in the theory, extracting events from sentences was one challenge we faced. We made the assumption that all sentences had a simple statement structure, were composed of either one or two events, and that the phrases describing each event were separated by a subordinating conjunction or some other part of speech such that

In detail, we first sought to find sentence phrases such as S or SBAR tagged subtrees within a Stanford parse of the original sentence. If no S or SBAR was found, the sentence was assumed to be a sentence containing only one event

Becky got a new puppy.

If the sentence did contain a sentence phrase of some sort, this was taken as a simple indication that the sentence contained 2 events (remember, we do not consider sentences with 3 or more events because of our simplifying assumptions). Since S and SBAR subtrees can be recursively nested, we chose the phrase highest in the parse tree (closest to the root). Then, we constructed the other event of the sentence, by creating the tree without the S/SBAR sentence phrase. For example, in the sentence:

Sarah traveled home to Cambridge after Alice was at the galleria.

the words “*after Alice was at the galleria*” are the leaves of the SBAR subtree closest to the root. We remove this subtree as one of the events of the sentence and manipulate the tree to create the other event composed of the full sentence parse tree minus the first found event subtree so we discover that “*Sarah traveled home to Cambridge*” was the other event in the sentence. The order in which the events occur in the sentence was preserved too, so that we could infer ordering by looking at clue words such as the subordinating conjunction “after”.

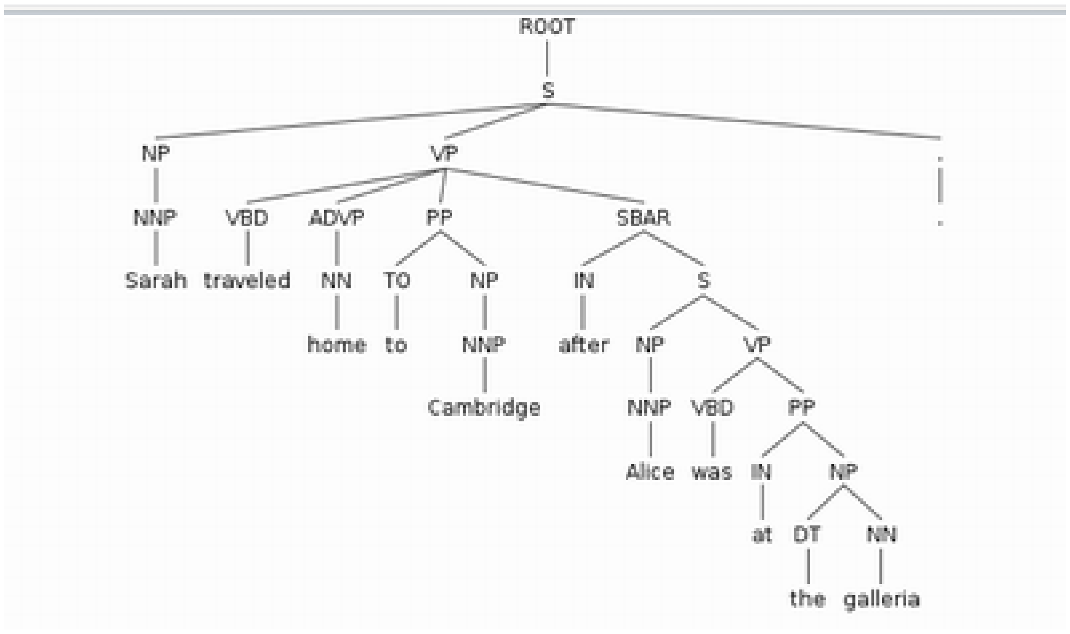


Figure 4

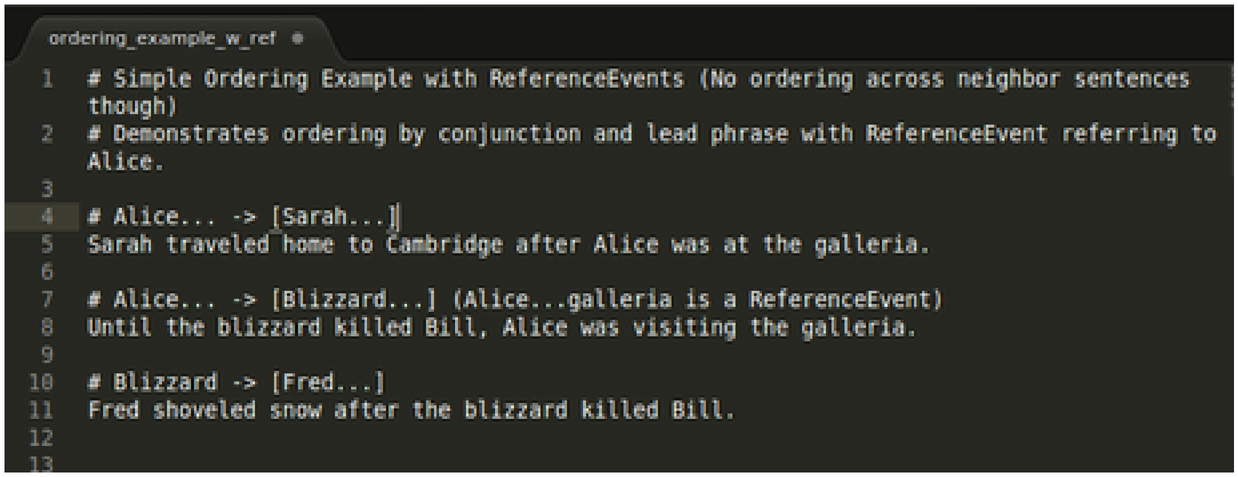
3.4 Event Ordering Analysis

3.4.1 Two Event Sentence Ordering

After extracting events from each sentence, the textual analyzer we built determines whether the sentence is a one event sentence or a two event sentence. If the sentence contains two events, the first three (or fewer if there are less than three) words of the first event are taken as the leading words of the sentence and the first three words of the second event are taken as the conjunction clue words.

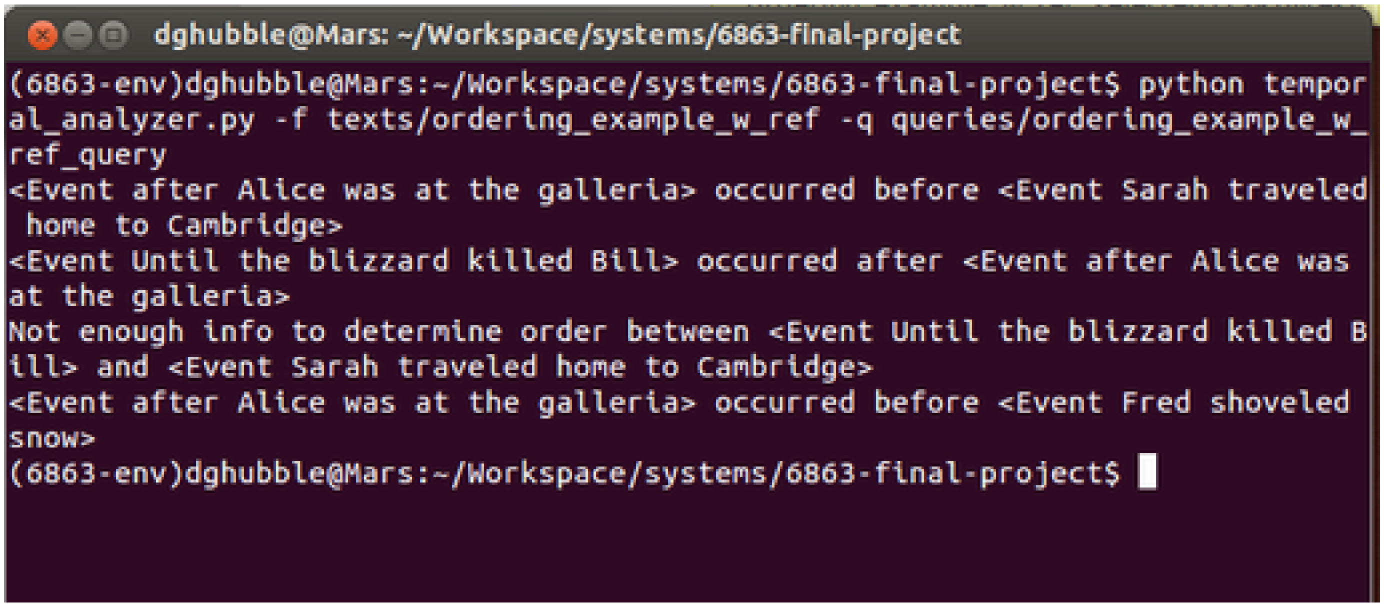
The leading clue words were compared against the chronological leading words catalog and the anti-chronological words catalog in order to score the two catalogs. Similarly, the conjunction clue words were compared against the chronological conjunction words catalog and the anti-chronological words catalog to score those catalogs. The scored catalogs were all compared to determine whether the chronological ordering or anti-chronological ordering was appropriate. If no catalog was the clear winner, no ordering was assumed.

Here is an example of a simple ordering among sentences with two events per sentence.



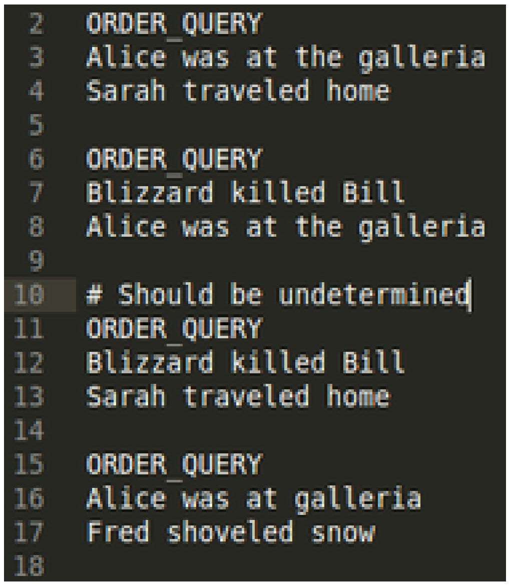
```
# Simple Ordering Example with ReferenceEvents (No ordering across neighbor sentences though)
# Demonstrates ordering by conjunction and lead phrase with ReferenceEvent referring to Alice.

# Alice... -> [Sarah...]
Sarah traveled home to Cambridge after Alice was at the galleria.

# Alice... -> [Blizzard...] (Alice...galleria is a ReferenceEvent)
Until the blizzard killed Bill, Alice was visiting the galleria.

# Blizzard -> [Fred...]
Fred shoveled snow after the blizzard killed Bill.
```

Figure 5: Note that the comments indicate the correct ordering we expect the system to learn from the sentence. The system ignores all comments (i.e. its not cheating here).



```
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ python tempor
al_analyzer.py -f texts/ordering_example_w_ref -q queries/ordering_example_w_
ref_query
<Event after Alice was at the galleria> occurred before <Event Sarah traveled
 home to Cambridge>
<Event Until the blizzard killed Bill> occurred after <Event after Alice was
at the galleria>
Not enough info to determine order between <Event Until the blizzard killed B
ill> and <Event Sarah traveled home to Cambridge>
<Event after Alice was at the galleria> occurred before <Event Fred shoveled
snow>
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ 
```

Figure 6: Textual analyzer runs some example queries and is able to correctly determine which events it knows an ordering about and which it does not have information about. Notice that the system is able to find the relationship between Alice and Fred, although they are not directly related.



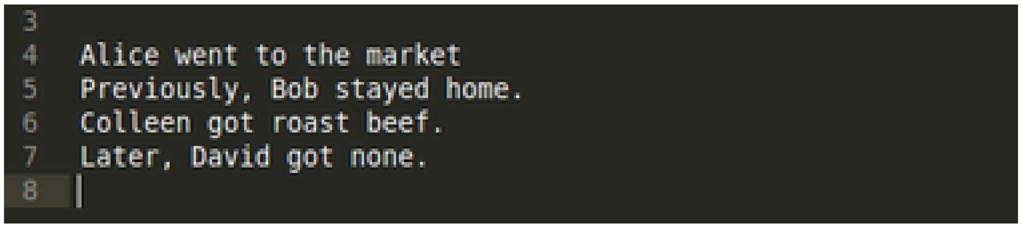
```
ORDER_QUERY
Alice was at the galleria
Sarah traveled home

ORDER_QUERY
Blizzard killed Bill
Alice was at the galleria

# Should be undetermined
ORDER_QUERY
Blizzard killed Bill
Sarah traveled home

ORDER_QUERY
Alice was at galleria
Fred shoveled snow

```

Figure 7: This is the query file which specifies the queries performed in the last example.

3.4.2 Single Event Sentence Ordering

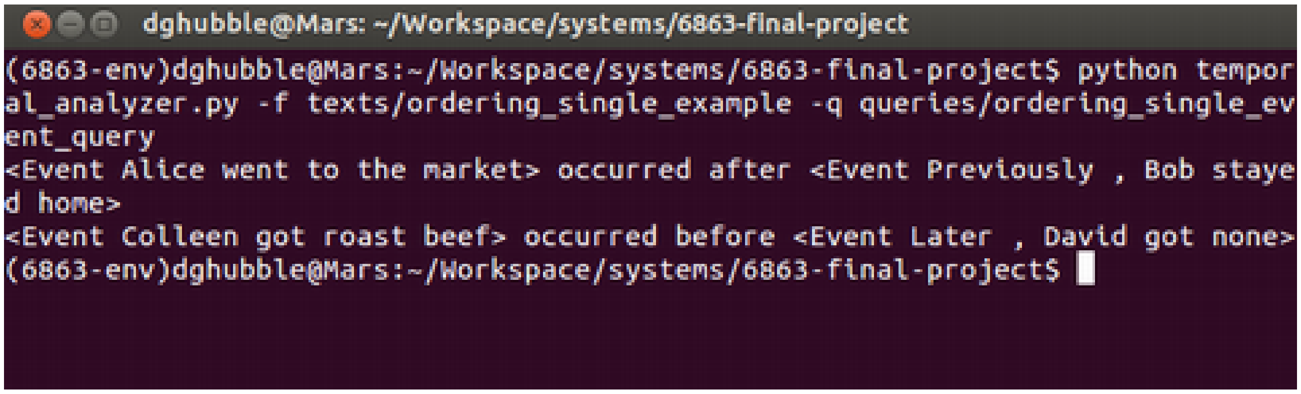
As discussed in the concepts section, orderings can also be created between the events of adjacent single event sentences. This is done by passing along a mono_prev_event variable to indicate the previous event from a mono event sentence (after analyzing a two event sentence will store None in mono_prev_event). Then, the leading three words of the event are taken and considered against the chronological and anti-chronological “across sentence” catalogs. Note that these catalogs are different from the 4 catalogs used in 3.4.1. These catalogs include words like “previously”, “later”, etc.

Here is an example of our system processing single event sentences:



```

Alice went to the market
Previously, Bob stayed home.
Colleen got roast beef.
Later, David got none.

```

Figure 8



```
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ python tempor
al_analyzer.py -f texts/ordering_single_example -q queries/ordering_single_ev
ent_query
<Event Alice went to the market> occurred after <Event Previously , Bob staye
d home>
<Event Colleen got roast beef> occurred before <Event Later , David got none>
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$
```

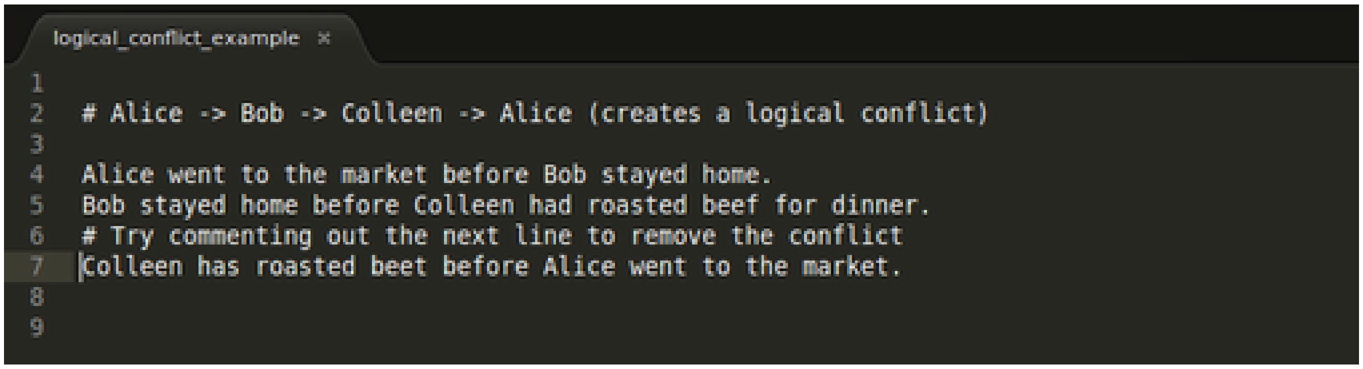
Figure 9

3.4.3 Non-Ordering

If an ordering cannot be conclusively made, no ordering is inferred. This simply occurs when neither the anti-chronological and chronological catalogs are dominant in scoring. Non-ordering is the default relationship among any events processed by the system.

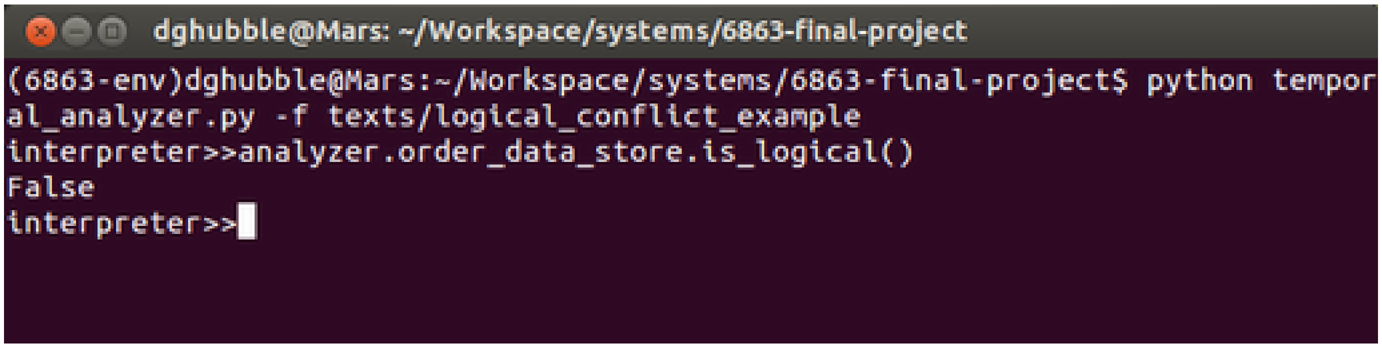
3.4.4 Logical Conflict Detection

It is possible for a text to describe an impossible ordering scenario which does not become clear until the entire text has been processed. For example, eventA may come before eventB. eventB comes before eventC, and then eventC occurs before eventA. This is clearly an impossible ordering situation. Our system used a depth first search over the ordering relationship datastore to find all cyclic relationships of this nature that are less than 10 relations in length. This functionality was made available through the *is_logical* method which returns a boolean of whether or not the processed text is logically consistent. If a cyclic ordering relationship was found, this is a logical conflict and *is_logical* will return False.



```

# Alice -> Bob -> Colleen -> Alice (creates a logical conflict)

Alice went to the market before Bob stayed home.
Bob stayed home before Colleen had roasted beef for dinner.
# Try commenting out the next line to remove the conflict
Colleen has roasted beet before Alice went to the market.
```

Figure 10



```
dghubble@Mars: ~/Workspace/systems/6863-final-project
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ python tempor
al_analyzer.py -f texts/logical_conflict_example
interpreter>>analyzer.order_data_store.is_logical()
False
interpreter>>
```

Figure 11

3.5 Event Time Estimation

One of the most basic features of the system is the ability to recognize time entities in text and extract them into useful information associated with an event. If an absolute time could be extracted from the text for each event, then it would be very easy to query the system to determine when an event occurred and whether one event happened before, after, or at the same time as another event. While not all (or even most) events have an exact date or time referenced in their text, it can still be very helpful to have absolute times associated with some events so they can act as a kind of “anchor” for all other events. For example, maybe you know the exact time (T) of Event A and you also know that Event B happened after Event A, so now you can tell that Event B happened some time after time T.

To go about extracting time entities from text, we started by looking at the code in the **timex.py** module from the **nltk_contrib** portion of NLTK code. While we did not end up using much of the code from timex.py, it was the basis for the solution we used for extracting this kind of temporal information. The basic approach is to use regular expressions to match patterns of text associated with different types of date and time information. The datetimes extracted from text are made up of combinations of year, month, day, hour, and minute values, and any combination of these can be present to constitute a date or time value. An example of time information that would be extracted from the text is something of the form “January 1, 1970”.

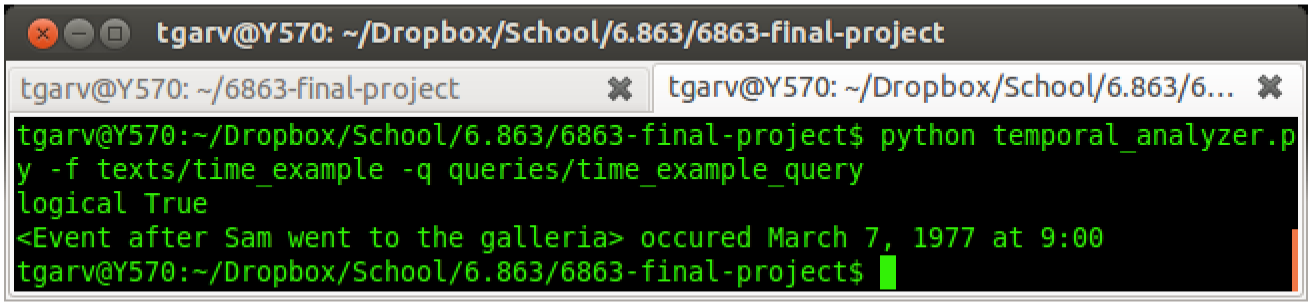
Once the time information has been matched using a regular expression, it needs to be stored in an object that makes that data useful to the system. The standard representation used for this in Python is **datetime**. The issue with using this representation for our system, however, is that a datetime needs to represent an exact moment in time and thus needs a value for each of year, month, day, etc. As mentioned before, the time information extracted from the text may not have all of these fields, as in the case of something like “February 2003”, which lacks day, hour, and minute information. A datetime object would not be able to capture the idea that the time refers to the entire month of February 2003, so instead we created a new class to represent this information. With the new Date class, it’s possible to have a representation of something with different degrees of specificity, but this also leads to issues when comparing two Date objects in which it’s not clear which one happened before the other.

When comparing Date objects, it's easy to tell that "February 10, 2003" happens after "January 15, 2003". But, what about the case where we want to compare "February 10, 2003" and just "February, 2003"? In this case, it's ambiguous as to which event actually happened first because they actually kind of occur at the same time. In this case, the comparison of Date objects will return a statement of ambiguity and admit that the system cannot actually determine which Date happened first, something that is not possible with a standard datetime object.

3.5.1 ReferenceEvent Time Propagation

In some cases, the text of a ReferenceEvent may contain a date that is more specific than the date already associated with the Event it refers to. In this case, the more specific Date can be propagated to the referred-to Event from the ReferenceEvent, so that better estimates for the time of occurrence can continue to add to the data we have about an Event. We added a notion of **precision** to the Date object as a way to show that a date such as "June 2012" is not as specific (or precise) as "June 30, 2012". With this measure of precision, we can continue to improve the time we have associated with each Event. Every time a ReferenceEvent is seen, we attempt to improve the precision of the Date associated with the Event it refers to.

An example of this functionality in action can be seen in **texts/time_example**. Notice that the first time the Event "Sam went to the galleria" was mentioned, there was no time associated with it. However, in a later ReferenceEvent to that Event, a specific time was given and that time was able to propagate back to the Event it referred to, providing a better idea of when the Event happened.


tgarv@Y570: ~/Dropbox/School/6.863/6863-final-project
tgarv@Y570: ~/6863-final-project
tgarv@Y570: ~/Dropbox/School/6.863/6...

```
tgarv@Y570:~/Dropbox/School/6.863/6863-final-project$ python temporal_analyzer.p
y -f texts/time_example -q queries/time_example_query
logical True
<Event after Sam went to the galleria> occured March 7, 1977 at 9:00
tgarv@Y570:~/Dropbox/School/6.863/6863-final-project$
```

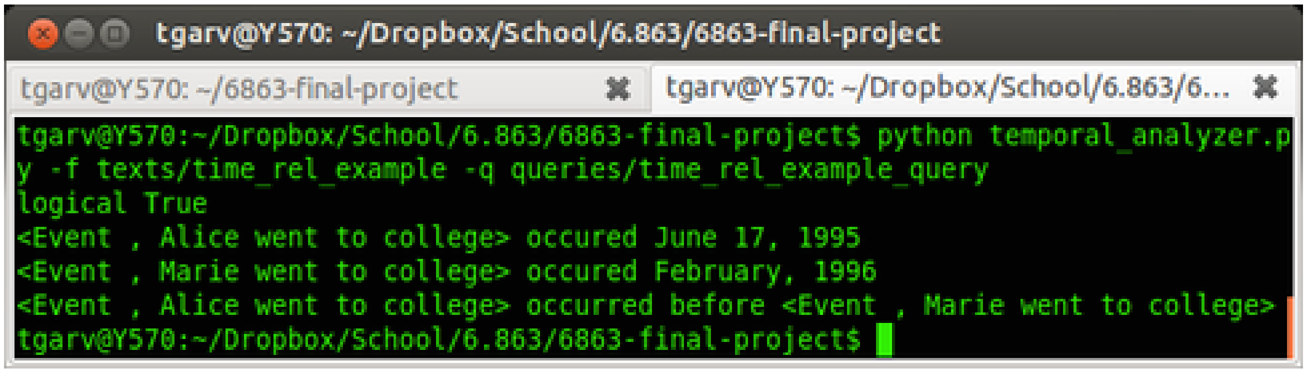
Figure 12: A screenshot of the terminal when running the example to propagate time information from a ReferenceEvent to the Event that it refers to. Notice that the query can determine when Sam went to the galleria even though that information was not mentioned at the time that the Event for "Sam went to the galleria" was created.

3.5.2 Grounding

Now that exact time information can be extracted by the system and associated with events, other interesting kinds of expressions can be processed and give even more information about when an event occurs. An example of this is seen in **texts/relative_time_example** with the sentence “Two weeks after Sam went to the grocery store, Alice went to college.” Since there is already an exact Date associated with the time that Sam went to the grocery store, it’s obviously possible to tell the exact time that Alice went to college.

To do this analysis, we use some of the code from **timex.py** again. Similarly to how times and dates were found, we use regular expressions to find text of the form [num][time period][before/after/earlier/later etc.] (e.g. “three months after”). **timex.py** has a very useful function called **ground** that takes one of these expressions and a Date object, and determines what time the expression actually refers to. In our example, ground can tell that “two weeks after” applied to the date June 3, 1995 (extracted from the event about Sam going to the grocery store) is referring to the date June 17, 1995. Now, our system uses this date and associates it with the event “Alice went to college”. With this combination of functionalities, interesting temporal information can be extracted from text referring to specific points in time.

For a full example of this kind of functionality, see the example text in **text/time_rel_example** and the associated queries in **queries/time_rel_example_query**. To run the demo for this example, run **`python temporal_analyzer.py -f texts/time_rel_example -q queries/time_rel_example_query`**


tgarv@Y570: ~/Dropbox/School/6.863/6863-final-project
tgarv@Y570: ~/6863-final-project
tgarv@Y570: ~/Dropbox/School/6.863/6...

```
tgarv@Y570:~/Dropbox/School/6.863/6863-final-project$ python temporal_analyzer.p
y -f texts/time_rel_example -q queries/time_rel_example_query
logical True
<Event , Alice went to college> occured June 17, 1995
<Event , Marie went to college> occured February, 1996
<Event , Alice went to college> occurred before <Event , Marie went to college>
tgarv@Y570:~/Dropbox/School/6.863/6863-final-project$
```

Figure 13: A screenshot of the terminal when running the simple demo to extract exact dates and relative dates. While the date June 17, 1995 was not explicitly stated as the time that Alice went to college, that information can be extracted from the text.

4. Usage

Please see the extensive README for installation instructions.

4.1 Getting Started

As stated, the system is accessed by invoking temporal_analyzer.py from the command line and passing different command line arguments. The first arguments to know about are **-f** and **-l**. **-f** is used to specify the textual file that should be read in to the analyzer. Remember, the analyzer reads sentence by sentence and learns all it can about events, their times, and their orderings in order to create its datastores which can be queried. All the example texts are in the texts/ folder of the problem.

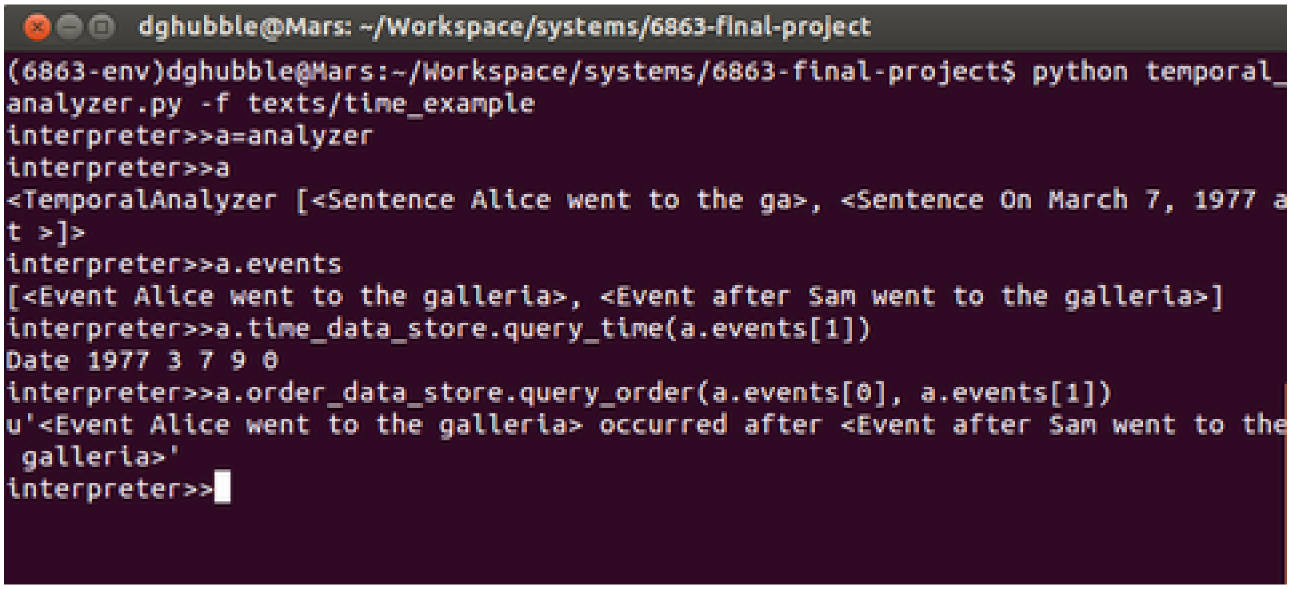
For example, you can run
```
python temporal_analyzer.py -f texts/time_example
```

After the analyzer finishes processing a text, so that you don't have to reprocess the text again next time, it automatically shelves the analyzer object that was created. This basically creates a persistent Python object stored to a file that can be loaded later very quickly. By default loading a file like texts/time_example will automatically save the shelved analyzer after doing text processing to the file shelved_data/time_example. Thus to load in the analyzer very quickly next time, use the **-l** option instead of the **-f** option.

```
python temporal_analyzer.py -l time_example
```

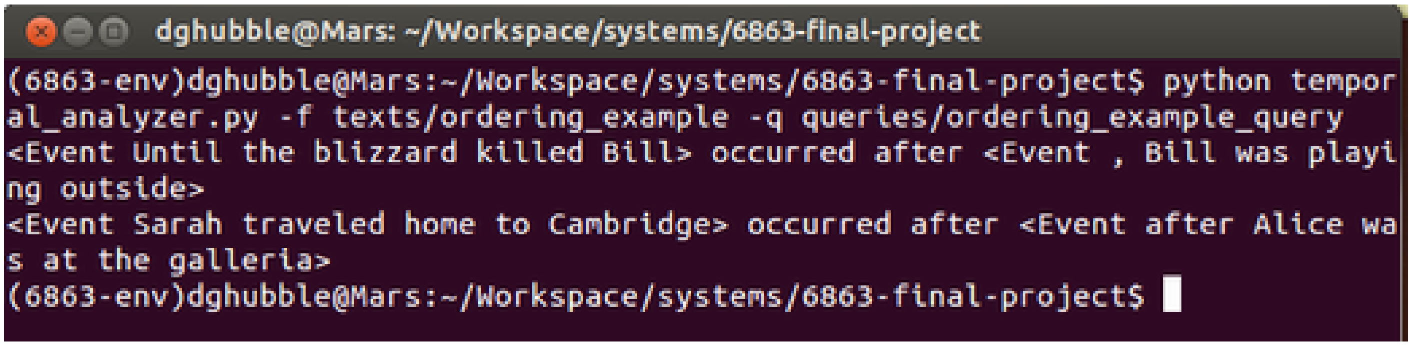
What has been described is **bootstrapping mode** which is all about getting to the state where the analyzer has learned about a particular text file, has its datastores populated, and is ready for you to perform queries on it.

The next and last mode to consider is **analyze mode** where you can analyze what the analyzer discovered about the text. If you ran either of the above 2 commands, you probably notice that you were presented by an interactive prompt. By default, after bootstrapping and shelving the analyzer, the program enters an interactive shell state. From this state, the analyzer is available with the variablename "analyzer". From here you can manually perform queries.


```
dghubble@Mars: ~/Workspace/systems/6863-final-project
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ python temporal_
analyzer.py -f texts/time_example
interpreter>>a=analyzer
interpreter>>a
<TemporalAnalyzer [<Sentence Alice went to the ga>, <Sentence On March 7, 1977 a
t >]>
interpreter>>a.events
[<Event Alice went to the galleria>, <Event after Sam went to the galleria>]
interpreter>>a.time_data_store.query_time(a.events[1])
Date 1977 3 7 9 0
interpreter>>a.order_data_store.query_order(a.events[0], a.events[1])
u'<Event Alice went to the galleria> occurred after <Event after Sam went to the
 galleria>'
interpreter>>
```
Figure 14

Alternately, you can also add the **-q** to specify a query file which contains all the queries you wish to run and they will automatically be executed. This is the recommended way of querying. A number of example query files corresponding to the example texts are available in the folder queries/.

```
python temporal_analyzer.py -f texts/ordering_example -q queries/
ordering_example_query
```

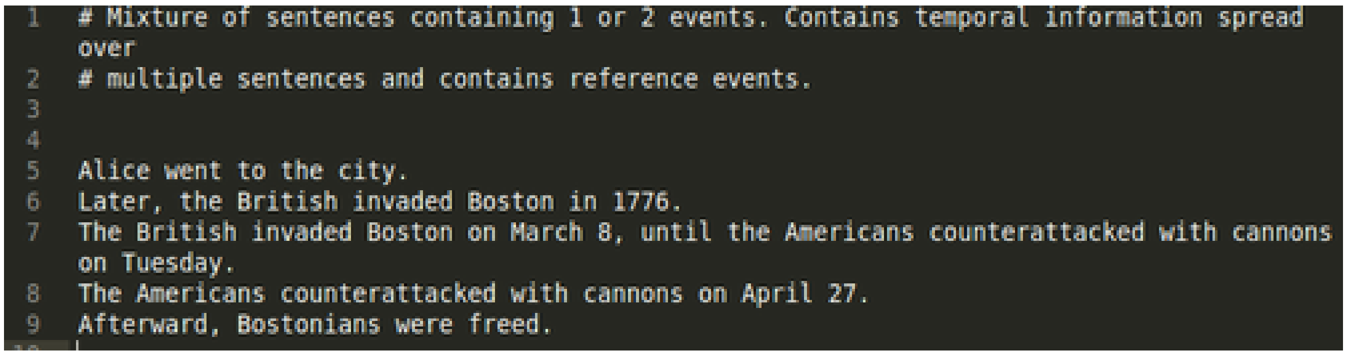


```
dghubble@Mars: ~/Workspace/systems/6863-final-project
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$ python tempor
al_analyzer.py -f texts/ordering_example -q queries/ordering_example_query
<Event Until the blizzard killed Bill> occurred after <Event , Bill was playi
ng outside>
<Event Sarah traveled home to Cambridge> occurred after <Event after Alice wa
s at the galleria>
(6863-env)dghubble@Mars:~/Workspace/systems/6863-final-project$
```

Figure 15

4.2 File Formats

The texts that the system processes should be in a format where each line is on a separate line and separated by a newline from the next. This file may contain comments beginning with the # character. An example text file is:



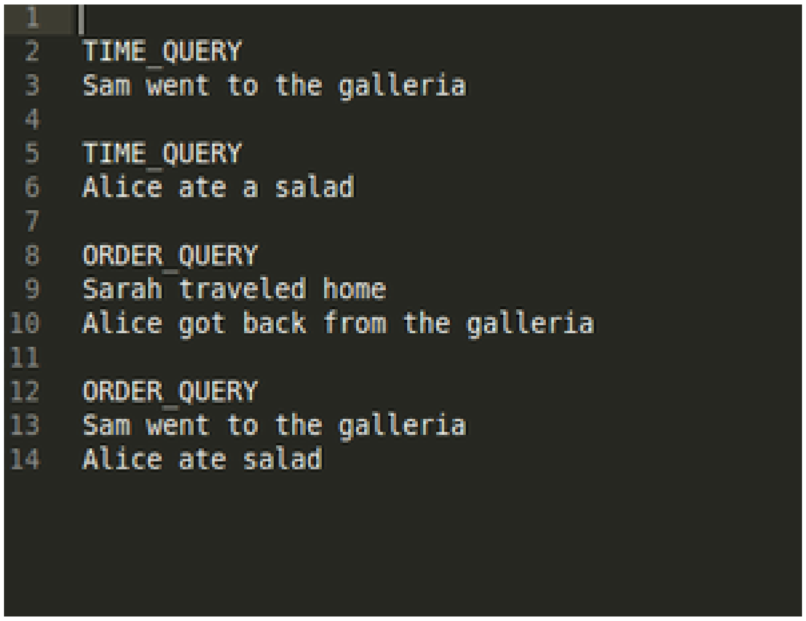
```
# Mixture of sentences containing 1 or 2 events. Contains temporal information spread
over
# multiple sentences and contains reference events.


Alice went to the city.
Later, the British invaded Boston in 1776.
The British invaded Boston on March 8, until the Americans counterattacked with cannons
on Tuesday.
The Americans counterattacked with cannons on April 27.
Afterward, Bostonians were freed.
```

Figure 16

The query file is a simple text file which uses the keywords TIME_QUERY and ORDER_QUERY to specify a query to be performed. You can list a sequence of queries in the same file. On a newline below TIME_QUERY, provide a string that is similar to the event you wish to get the time of and the corresponding Event will be found and the best available merged time information about that event will be returned.
ORDER_QUERY, naturally expects two arguments so on the next line after ORDER_QUERY place the description of the first Event and on the next newline, place the description of the

second event. The query will be performed and will provide you with information about the order in which the two described events occurred.

An example Query File is:



```

TIME_QUERY
Sam went to the galleria

TIME_QUERY
Alice ate a salad

ORDER_QUERY
Sarah traveled home
Alice got back from the galleria

ORDER_QUERY
Sam went to the galleria
Alice ate salad
```

Figure 17

5. Future Work

This project contained numerous difficult subproblems and challenges which had to be overcome with imperfect solutions and simplifying assumptions. To further improve this project, there are several areas that could be developed. The first difficult subproblem was sentence parsing in order to perform event extraction. While we made the assumption that all sentences contain at most two events and that events are separated by subordinating conjunctions, this assumption greatly limits the variety of sentences that our system is able to correctly process. A well designed event extraction module could be incorporated into this project to improve its overall operation without changing the way ordering and timing are inferred. The second subproblem hindering the functionality of our system is the difficulty in matching strings to determine whether a particular event refers to a previously seen event, i.e. whether an event is a ReferenceEvent or not. Again, a well designed string matching function could be substituted into our system with little effort and could greatly improve performance in this area.

In the future we would like to expand the breadth of cataloged ordering clue words and how they are extracted from sentences. In this system, the ordering words in the catalogs are fairly simple and clue words are drawn from the leading words of a sentence and the beginning of conjunction phrases. This simple approach limits the variety of the events we can order so we could introduce new strategies for inferring event orderings to deal with more general, non-simplistic sentences.

We would also like to incorporate the notion of a “current” time of the events mentioned in the text into the online sentence processor so that relative words like “now”, “earlier”, "currently" can provide richer timing and ordering information. This would also allow words like "today" and "tomorrow" to have meaning that is not recognized in our system now. This would be especially useful for processing texts written about events going on presently in the speaker's context such as in news article reports.

6. Conclusions

We learned quite a bit about the difficulties of building truly general natural language processing systems, discovered simplifying assumptions that allowed us to scale down a very difficult NLP problem (which included numerous challenges that are difficult NLP problems in their own right) to make it tractable, and wrote some pretty good software demonstrating that computers systems can learn quite a bit about events, the timing of different events, and the relative ordering of different events.

These capabilities enable useful functionalities such as querying about the best available time at which an event occurred, even if that information was originally spread across the text. Queries can also be made to determine the relative ordering of events described in a text, even if the temporal relations connecting events only become clear from processing the entire text because information is spread throughout the text.

Capabilities like these potentially save humans time and demonstrate that computer systems can extract the essential materials out of written texts to the degree that they can even detect logical conflicts within a text due to cyclic relative event orderings.