Dependency Parsing

Summary

For this coursework, the technique of dependency parsing in reference to extracting opinions on Amazon DVD reviews is experimented with, analysed and commented on. Using the output of the parser, the dependencies are analysed algorithmically to produce an opinion extractor capable of labeling a sequence of text with appropriate opinions expressed within. The work is broken down into 3 main sections, an implementation, analysis of the extractors performance, and a further comment on how one might productionize this process in the form of a website that summaries opinions on DVD’s

Implementation

The extractor works using a layered pipeline that starts with an input of text, and performs systematic processing at various stages of the pipeline before outputting a list of string that’s represent opinions based on that text. The pipeline can be abstracted using the following depiction:



Figure Opinion Extractor Pipeline

Raw text enters the system and is first sent to the tokentizer. This breaks the string into a series of rudimentary tokens. Each token roughly equates to a word in the English language with some exceptions - such as breaking ‘wasn’t’ into two tokens [‘was’ and ‘nt’]. The actual implementation used in this system is the Treebank regular expression tokenizer[1] provided in the NLTK library. The tokens are then supplied to a part of speech extractor that divides each token into tag classes that behave similarly. Nouns, verbs, prepositions are examples of such tags. The NLTK library again uses the Penn Treebank tagsets[2] to provide this classification. These tokens are then fed into the dependency-parsing phase of the pipeline. There are two main ways of achieving dependency-parsing – graph based and transition based models. The implementation used in this project utilizes the later approach by greedily looking at all tokens and corresponding POS tags and greedily assigning certain ‘transitions’ between different words. Decisions on what transition should be assigned is determined a classifier such as a support vector machine trained against a corpus of training data. The classifier then uses this training set to build a model of most appropriate transition decisions. This implementation uses the Stanford dependency parser based on the Penn Treebank POS tags[3]. The output of this stage is a directed tree representing the dependency transitions between the tokens. This structure is then supplied into a custom opinion extractor to isolate the opinion sentiments by traversing the tree. The extractor is described in the below pseudocode.

Input:

queryToken: the token we’re interested in. It is part of the dependency tree of tokens.

Output:

opinions: an array of token forms (which constitute the extracted opinions related to the query token).

**function** opinion\_extractor*:*

allOpinions =[]

opinions = []

**if** head of queryToken has a ‘conj’ relationship to queryToken, then:

allOpinions = **call** opinion\_extractor(queryToken.head)

adjectives = []

**for each** dependent token in queryToken’s set of dependents:

**if** the dependency relation of the dependent = ‘amod’ or

the head of queryToken has pos starting with ‘JJ’ or ‘VBG’

and a relationship of ‘nsubj’ to queryToken, **then**:

**append** a new BaseOpinion to adjectives

**for each** adjective in adjectives

adjective = **call** getPrepositionModifiers(adjective)

**append** adjectives to opinions

**for each** opinion in opinions

**for each** result in **call** getAdverbialModifiedOpinions(opinion):

**append** result to opinions

**if** head of queryToken has pos tag starting with ‘VB’ except ‘VBZ’ and

a relationship of ‘dobj’ to queryToken, **then**:

**append** a new BaseOpinion to opinions

**if any** dependents of opinion’s token have a relationship of ‘neg’

**then** decorate all opinions with a new NegateOpinion

**for each** opinion in opinions:

opinion = **call** negateOpinion(opinion)

**for each** opinion in opinions:

conjunctions = []

**for each** dependency of opinions rootToken:

**if** dependency has a relation of ‘conj’ then:

append a new BaseOpinion to conjunctions

**for each** conjunction in conjuctions:

**for each** result in

**call** getAdverbialModifiedOpinions(conjunction):

**append** result to conjunctions

conjunction = **call** negateOpinion(conjunction)

**append** conjunctions to opinions

return allOpinions appended with opinions;

Input:

opinion: the opinion we would like to see if adverbial modifications can be applied

Output:

opinions: a list of opinions that may or may not have been decorated as a AdverbialModifiedOpinion. A single opinion as input can be split into multiple opnions, each decorated with a different adverb

**function** getAdverbialModifiedOpinions*:*

admods = []

opinions = []

**for each** dependency of the opinion’s rootToken:

**if** dependency has a relation of ‘advmod’, then:

append dependency to admods

**if** length of admods > 0 then:

**for each** admod in admods:

**append** new AdverbialModifiedOpinion(admod, opinion)

decorator to admods

**else**

**append** opinion to opinions

**return** opinions

Input:

opinion: the opinion we would like to see if negation modifications can be applied

Output:

opinion: an opinion that may or may not have been decorated as a NegatedOpinion.

**function** negateOpinion:

**for each** dependency of the opinion’s rootToken:

**if** dependency has a relation of ‘neg’, then:

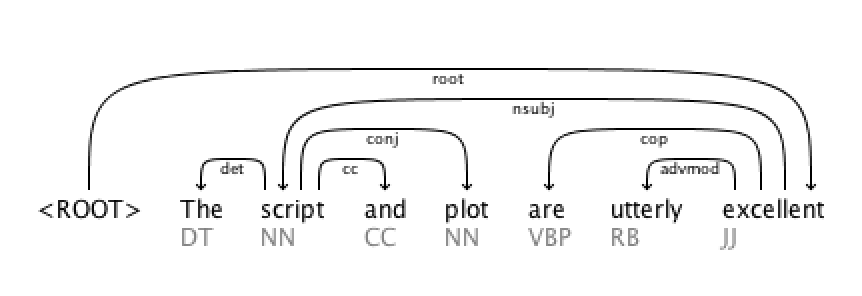
**return** new NegateOpinion(opinion) decorator

return opinion;

There are some key points about the algorithm that require further description. Firstly, a series of datastructures have been created to capture different opinions. These datastructures all follow the decorator pattern and allow complex chaining of sentiments to be tracked and recalled. Every opinion has a ‘rootToken’ that represents the core property of the opinion such as ‘fun, excellent etc. Different implementations of Opinion will either return the rootToken directly or, if they are a modifier opinion, they will delegate to their decorated opinion. When looking to obtain the actual form the opinion takes, the delegation pattern again builds a string based on the chained opinions and their modifications by traversing the object graph. There are 6 Opinion objects:

|  |  |  |
| --- | --- | --- |
| Opinion Object | Purpose | Arguments (object graph descedents) |
| BaseOpinion | This stores a base opinion such as an adjective describing the queryToken | Just the BasicToken describing the relationship to the queryToken |
| AdverbialModifiedOpinion | This represents an opinion that has been modified with an adverb | Takes a BaseOpinion as it’s direct descendent along with the adverb BasicToken that describes the modification |
| NegatedOpinion | This negates any opinion | Takes any opinion as its direct descendent |
| PrepositionalModifiedOpinion | This behaves as a link between prepositional token creating a ‘super’ opinion from a number of tokens all chained together | Takes a BaseOpinion as a direct descendent and a BasicToken denoting the preposition |
| PrepositionalObjectModifiedOpinion | Represents the object that is the actual object of the preposition (pobj) | Takes a PrepositionModifiedOpinion object as it’s direct descendent and a BasicToken detailing the pobj relation |

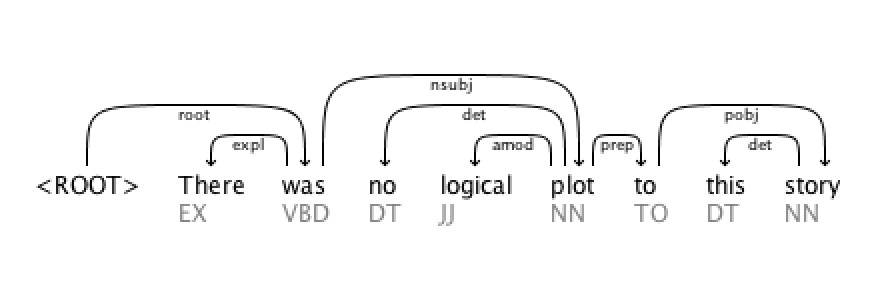
At the beginning of the routine, the algorithm also uses recursive calls to obtain backward-chained conjunctions such as in the example:



Here, the query token is ‘plot’ but we can see that the adjective ‘excellent’ is being applied to both script and plot. In order to locate this relationship, we need to trace back to the root noun and apply the adjectives from this node in the tree. The algorithm performs a recursive search to ensure all conjunctions are traced (such as ‘the characters and script and plot are utterly excellent’). It is also worth noting that the python implementation of the above routine abstracts a lot of the code into modularized methods for clarity, reuse and maintenance.

Analysis:

The opinion extractor was first built using a suite of unit tests that test the extraction process by applying the test sentences in the lab material to the extractor and observing the result. These can be run and verified that all pass as per the requirement. The only issue is with the fifth extension sentence: “There was no logical plot to this story”. The dependency relations for this sentence is:

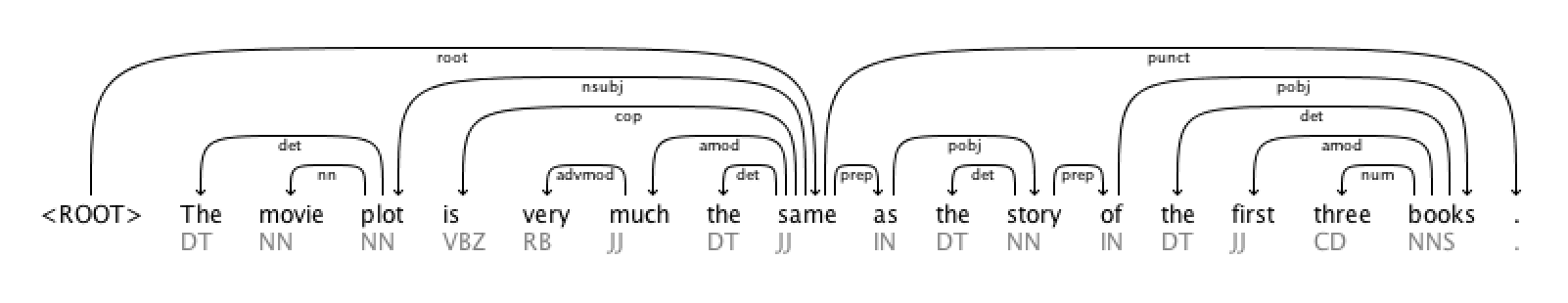


Here it can be observed that the problem is not with the opinion extractor. The problem is that the ‘no’ token before logical is not labeled from the dependency parser as ‘neg’ but instead a determiner (‘det’). This makes it impossible for the opinion extractor to infer that it should have negated the opinion. Brittle code could have been added to look for this particular word rather then at the relationship but the impact of such a change is difficult to predict to unseen sentences and unwanted regressions could be introduced. The best approach to tackle this issue is, rather then to patch the problem at the extractor stage of the pipeline, fix the dependency parser to correctly label the negation relationship. This was deemed out of scope for this project.

After the core of the opinion extractor was tested, its performance was observed against less formulated examples such as the Amazon DVD review corpus. The following is a description of successful examples and ones that failed to obtain the correct sentiment and an analysis of where the failures occurred.

Successful Extractions:

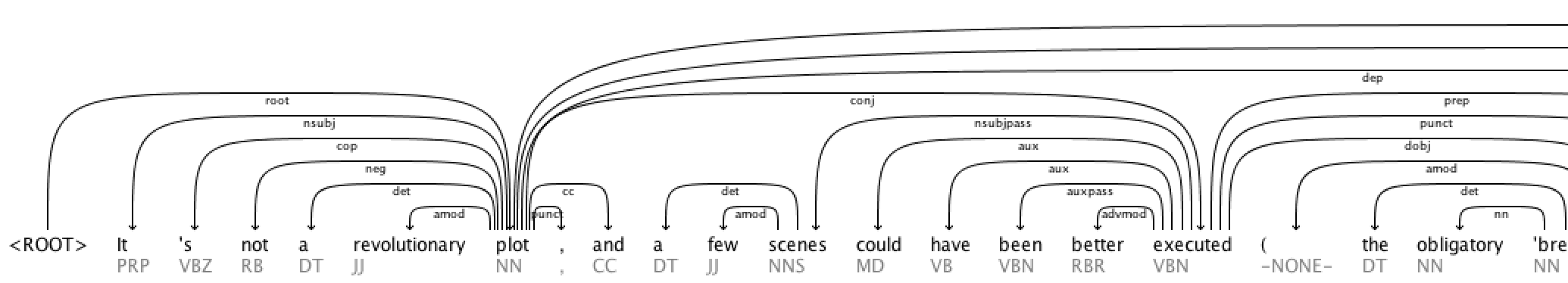
*“The movie plot is very much the same as the story of the first three books”*

*queryToken = ‘plot’*

*extracted opinions: ['same-as-story-of-books']*

When looking at the opinion of the word ‘plot’ in the above sentence, we can see that a single opinion has been returned from the extractor. This opinion pretty much summarizes the sentiment of the review in relation to the plot. A possible improvement would be to include the ‘first-three’ tokens in the opinion so that it can be determined that the plot is the same as only the first three books. This extension could be accomplished in the extractor by adding a modifier to deal with the ‘num’ dependency. Adding code to look at the PrepositionalObjectModifiedOpinion opinion and call the getAdjectiveOpinions method could also obtain the ‘first’ adjective from the statement.

*“It 's not a revolutionary plot , and a few scenes could have been better executed ( the obligatory 'break-up ' before the happy ending is rather weak ) , but on the whole the mushy side of me ate it up .”*

**

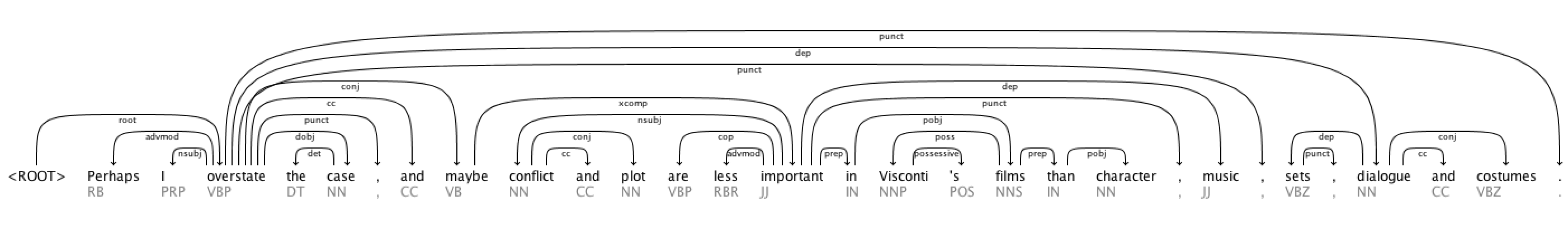
*queryToken = ‘plot’*

*extracted opinions: [‘not-revolutionary']*

This appears to be completely accurate with no obvious possible improvements to comment on.

Failures

*“Perhaps I overstate the case , and maybe conflict and plot are less important in Visconti 's films than character , music , sets , dialogue and costumes .”*

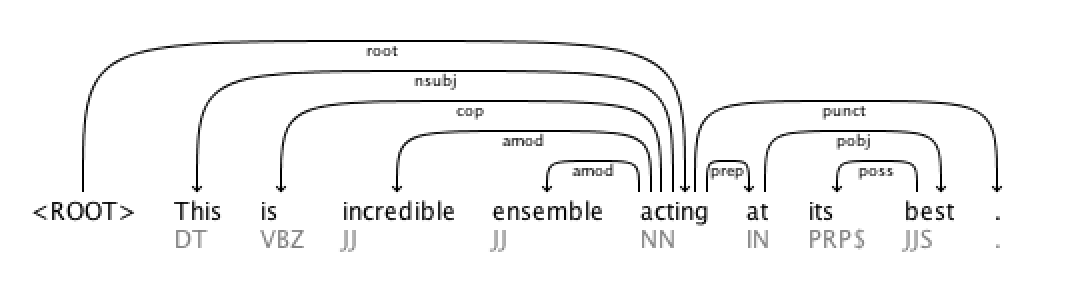


*queryToken= plot*

*extracted opinions: ['less-important-in-films-than-character']*

This extraction appears to be pretty accurate, correctly determine that this article has the opinion that the plot is less important in films than characters. The problem of course is that the opinion here is slightly wrong. It is Visconti’s films that the plot is less important than. This could be fairly easily determined by including modifiers for the possession dependency, denoting that it is Visconti that has possession of the films being referred to here. Also it is not just the characters that are superior to the plot but also the music, sets, dialogue and costumes. This is down to the fact that the dependency parser has failed to conclude that a list of commas followed by an ‘and’ should mean that all tokens be linked by conjunctions. Without these relations, the opinion extractor is unable to determine that character is linked to the other nouns.

*“This is incredible ensemble acting at its best .”*



*queryToken = ‘acting’*

*extracted opinions: [‘incredible', 'ensemble']*

Here the extractor has return two opinions ‘Incredible’ and ‘ensemble’. Really this is one opinion that the acting was an ‘incredible-ensemble’ as it is a adverb modification of incredible on the adjective ensemble. Again though the issue appears to be due to the fact that the dependency parser has determined two distinct relationships of incredible and ensemble to the acting query token. Really this should have been determined an advmod relationship between incredible and ensemble.

Biblography

[1] Penn Treebank Tokenizer - <http://www.cis.upenn.edu/~treebank/tokenization.html>

[2] Penn Treebank POS tagset - <http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html>

[3] Stanford typed Dependency - <http://nlp.stanford.edu/downloads/dependencies_manual.pdf>