Thesis

Ryan Tanner

February 26, 2012

Contents

1 Introduction

2 Problem Statement

This thesis is an attempt to tackle the problem of extracting facts and connections from written text. Massive quantities of text are produced daily and methods for quickly getting relevant information out of that text are needed. There are many

2.1 Extracting properties defined in a massively large body of text

2.2 Doing so in a time-efficient manner

2.3 On a textual level, the problem of finding connections across concepts and entities in a massive corpora of text

Creating a property map for a single, discrete document is relatively easy using the algorithm described in this paper. The larger challenge is connecting entities across documents.

2.4 Motivation

This project can be divided into two broad sections and their motivations will be dealt with separately.

First, the language processing component. The use of dependency grammar over the more widely-used constituent grammar was motivated by a desire to test the possibility of applying functional (programming) language concepts to natural language. More colloquially, I wanted to scratch an intellectual itch. Treating grammatical functions as computation functions and modeling them as such appealed to me as an elegant proposition. I freely admit that I had no basis for this approach. Building this project seemed the best way to test my hypothesis.

3 Other Approaches

3.1 Training oriented approaches

3.1.1 Manually-annotated training input

3.1.2 More accurate than my proposed solution

3.2 Weakly-linked crowd sourcing

4 Importance of the Problem

4.1 The Problem of Big Data

Google alone processes over twenty petabytes of data per day (**?**).

4.2 Tracing Influence

5 Approach to Solving the Problem

Most approaches to this problem rely on extracting as much information as possible from a given input. This approach comes at the problem from the opposite direction and tries to extract a little bit of information very quickly but over an extremely large input set. My hypothesis is that by doing so a large collection of texts can be quickly processed.

5.1 Treating grammatical dependencies as functions

This approach is based on the premise that dependency grammar relations can be treated as functions and modeled as such. Furthermore, I hypothesize that these functions can be curried, just as in a functional language. Every word in a sentence, save for the head, is dependent upon another word and each of these dependencies has a type. This structure forms a tree. By doing a depth-first traversal of this tree and recursively composing each individual dependency function into a curried function, we end with a function specific to that sentence.

In this approach, dependency functions are short operations which extract properties from the given relation. These functions take two nodes of a tree as input, the governor and the dependent. Based on the part-of-speech tags of the tokens in each node a partial or full property is added to the accumulator map and returned up the tree. This map is comprised of entities mapped to properties representing pieces of information extracted from the relationship. More about properties can be found in section 6.2 on page 5.

5.2 Mapping the governors and dependents of those dependencies to textual aliases and named entities

Using the coreference resolution functionality of the Stanford NLP library, properties and entities can be mapped together. In addition, this function of this library solves the problem of resolving ambiguous nouns like “it” or “they” by resolving them to a representative alias.

Within a document, entities are mapped to aliases by a simple inclusion-test. That is, entities (as defined by the parse tree algorithm) are mapped to aliases by testing if an entity’s tokens are contained by an alias. In essence this “smoothes out” the results of the parse tree algorithm and collapses the tree, providing a rough form of deduplication within a document. If an entity is contained within a non-representative alias, it mapped not to that immediate alias but to that alias’s representative form.

5.3 Reducing a set of input documents to find connections between those aliases and entities based on their common properties

In order to connect entities which appear in separate documents, a simple string matching algorithm has been devised which checks for the existence of another representative alias

5.4 Constructing a graph of these connections where the connections form weighted vertices and entities form nodes

5.5 Visualizing this graph

5.6 Why a functional language?

6 Algorithm in Detail

6.1 Dependency Functions

The grammar dependencies used here are those described in the Stanford typed dependencies manual **?**. Currently 53 grammatical relations are defined for the English language. Each of these has a corresponding function in this algorithm. Though the specifics of each function differ, all follow the same simple pattern. Dependency functions take two parameters, a governor and a dependent, and return a map of tokens to a list of properties. Furthermore, these grammatical relations have a typed hierarchy where relations can inherit from other relations. Each function therefore can use its supertype's own function and only add the minimum processing necessary for its specific relationship.

These functions use the part-of-speech tags of the tokens in the governor and dependent to loosely determine if a full or partial property is defined by this relationship.

These functions were arrived at by first considering their meaning in the context of English grammar and their definitions within the Stanford parser and then through a process of trial-and-error. An example follows:

```
class nsubj extends dep {
  override def apply(gov: ParseTreeNode, dep: ParseTreeNode):
      Map[Token,List[Property]] = {
    val props = super.apply(gov,dep)

    val nns = (new NounProperty(dep.word) /:
             (props.getOrElse(dep.word,List[Property]())
               filter { _.getClass == classOf[NounProperty] }
               map { np:Property => np match {
                   case np2: NounProperty => np2
                   case _ => throw new ClassCastException
               } }
             ) ) { (np:NounProperty,nn:NounProperty) => nn ++ np }

    val temp = add(props,
                  dep.word,
                  nns,
                  { p:Property => p.getClass == classOf[NounProperty] })

    return add(temp,gov.word,new Subject(nns))

  }
}
```

6.2 Properties

6.2.1 Full vs. Partial Properties

Given that a property may not be fully defined by a single grammatical relation, this solution provides for "partial properties" to be returned by dependency functions. This can be seen as a loose form of delayed message-passing. Dependency functions which begin a partial property add it to the map entry of the governor's token. Conversely, dependency functions which can use a partial property to create a full property check for the existence of such partial properties in the map returned by child nodes. As we use a depth-first traversal child nodes are processed before parent nodes, allowing partial properties to be in effect passed from child to parent and then filtered by the parent before returning the result map.

7 Input Data

Two principle sets of input data were used in large-scale testing of this algorithm: first, a collection of several thousand (get exact number...) articles from Wikipedia and second, the article on World War II and those immediately linked by it. The motivation behind this decision is two-fold:

In the first place, processing the entirety of Wikipedia is not practical without a great deal of manual "massaging" in order to get the input documents into an acceptable form. While the Wikimedia Foundation does provide dumps of their database, it is not perfect and correctly reassembling that database is a tricky process. Of the roughly 1.5 million articles contained in the dump used here, only about 250,000 were suitable for tagging. Of those, some contained malformed markup or other anomalies which rendered them unsuitable for tagging with the Stanford library. While the resulting set of documents is still large, it is by no means complete.

This problem led to the two-pronged approach of also testing with a small but closely-related set of documents: those known to be relevant to the subject of World War II. World War II was chosen as the basis for this test for several reasons. Firstly, it is a topic which spans (quite literally) the entire world, providing a robust test to the geolocation aspects of this project. Secondly, Wikipedia's body of text on the subject is sufficiently large to challenge the algorithm used here. Third, this topic is well-known to any reader or user and therefore allows the user to quickly see the usefulness and relevancy of the approach taken here.

8 Results

9 Future Recommendations

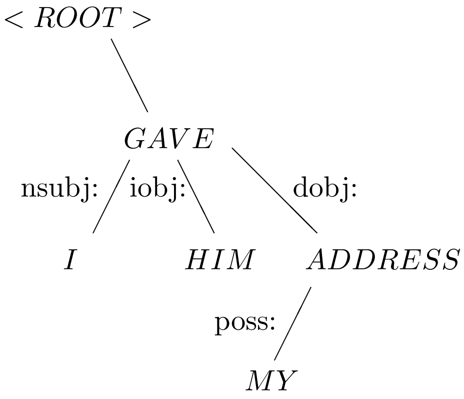
A Some Relevant NLP Concepts

A.1 Dependency Grammars

Dependency grammars are a class of grammatical theory not based on context-free grammars but instead on the relations between words. (Jurafsky, 354). This class of theory was first explored by early Greek and Indian linguists and regained popularity in the latter 20th century.

Dependency grammars are quite simple: every word in a sentence is dependent upon another word (save for the head or root word) and these relationships form a tree structure as seen in the figure below. Furthermore, each relationship has a type. This structure abstracts word order away, meaning that multiple sentences can map to the same dependency tree.

Various specific grammars exist for the English language; the grammar used here is that employed by the Stanford library which defines 55 semantic relations.



In this sample sentence, "I gave him my address," we can see how $GAVE$ is the root word of this sentence, that is it has no governor word. Every other word in the sentence is a dependent of another word and each of these dependencies has a type. The relationship between I and $GAVE$ is given as $nsubj(gave-2, I-1)$, indicating a *nominal subject* relationship between the tokens "gave" and "I" at indices 2 and 1, respectively.

A.2 Why dependency grammars?

Dependency grammars appealed to me for a number of reasons: functionalism, the simplicity of representing a sentence as a tree and their potential for property extraction. This form allows for verbs to be viewed as infix functions and identifies their arguments. Doing so helps quickly extract statements of the type "$Entity_i$ modifies $Entity_{i+1}$" and bind both these entities and the type of modification to a given instance. Doing so establishes relationships between entities and properties of those entities.

Essentially, dependency grammars help reduce complex sentences to less-complex relations, greatly simplifying the task of extracting these relationships from a document. While a tremendous amount of contextual information is lost, the tradeoff allows for a small set of relatively simple functions to extract information from a wide range of input documents.

B Tools Used

C Code Highlights