Graphical recipe automation

Kristinn Björgvin Árdal

May 15, 2015
BSc Discrete Mathematics and Computer Science

Author: Kristinn Björgvin Árdal
SSN: 081190-2169
Instructor: Sigurrós Soffía Kristinsdóttir
Supervisor: Hannes Pétursson
School of Computer Science
Reykjavík University

Abstract

Graphical recipes are a way of representing cooking instructions and we wanted to make the process of making them automatic. The software would be able to read in a recipe text, ingredient list and instructions, and output a graphical recipe. To extract information out of written text Natural Language Processing was used. We ran into some trouble with part of speech tagging the text and in the end trained our own tagger with an accuracy of 97%. The software does not generate perfect graphical recipes and some changes are needed until it is complete. The automatic conversion process is very quick and its purpose is to help people in making graphical recipes. Using Natural Language Processing we were able to generate graphical recipes that were on average 70-80% accurate.

Contents

1 Introduction 1
2 Methods 2
  2.1 Sentence segmentation 5
  2.2 Tokenization 5
  2.3 Part of Speech Tagging 5
  2.4 Entity detection 6
  2.5 Relation detection 8
  2.6 Ingredient list 9
  2.7 Sample run 9
3 Results 10
  3.1 Tagger Accuracy 10
  3.2 POS-tagging running time 11
  3.3 Accuracy 11
4 Discussion 12
5 Conclusion 13

1 Introduction

This project was done in collaboration with Gracipe (`www.gracipe.com`), a start-up company that is changing how food recipes are presented. Gracipe has created software for creating graphical food recipes. The process of making a graphical recipe consists of pasting into an online editor all the ingredients and tools necessary for completing the dish. Next the ingredients and tools are dragged and dropped into the workspace in the right order to indicated how the ingredients are mixed together and how the tools are used to make the dish. An example of a graphical recipe can be seen in figure 1.

The idea of this project is to make conversions of already existing recipes to graphical recipes easier. The process would then consist of pasting a whole recipe, ingredients and instructions, into an online editor and output a graphical food recipe. Next the user would be able to edit the graphical recipe and add comments if needed. The recipe input is to only be English and the conversion

Figure 1: Gin and tonic graphical recipe example

was not supposed to be perfect. The reason for the automation not being perfect is that the English language is ambiguous [1, ch. 8] and therefore there are cases that can be interpreted in more than one way.

It was decided that natural language processing was to be used to interpret the recipe instructions. The software was written in python and the "Natural Language Toolkit" module (NLTK) was used for processing the recipe. NLTK is open source software distributed under the terms of the Apache License Version 2.0. In natural language processing there are a few steps in extracting information from text. These are, sentence segmentation, tokenization, part of speech tagging, entity detection and relation detection as can be seen in figure 2. More detail on what these steps are and how they were implemented can be found in the section 2.

The project was done by Kristinn Björgvin Árdal with help from Kai Köhn, the CTO and Lead Developer at Gracipe.

The final look of the software on the website can be seen in figure 3. The look of the graphical recipe is not linked to the recipe conversion in any way and can be changed without changing the underlying software for the automation.

2 Methods

The software should be able to accept a the text for a recipe with both an ingredient list and instructions. The ingredient list would come first separated from the

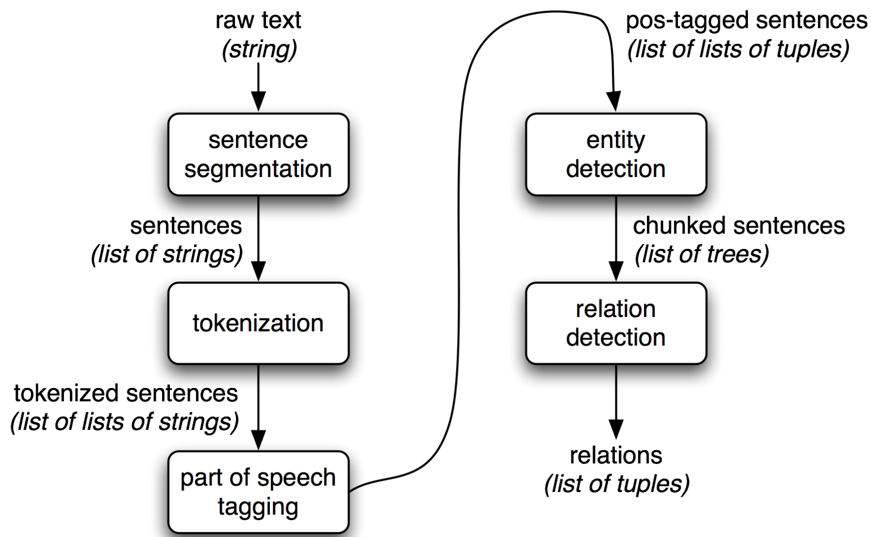

Figure 2: Pipeline for information extraction.

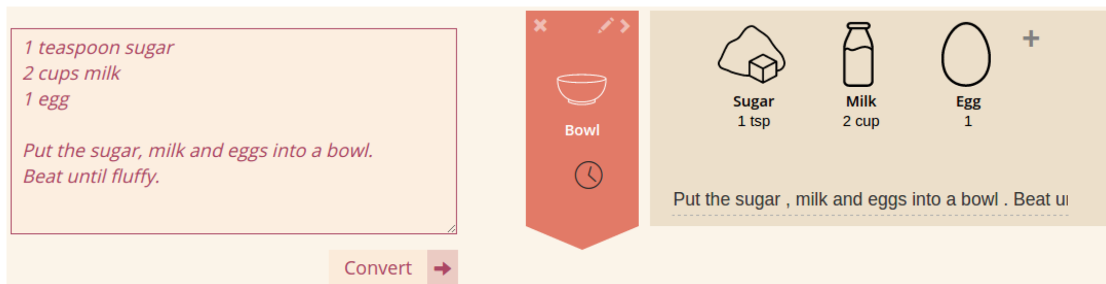

Figure 3: Look of graphical recipe editor on website.

instructions with an empty line. The software then go through the ingredient list and figure out what ingredients are used and the amount if specified. After that it parses the instructions finding relations between ingredients and tools and output an ordered list of steps. This list of steps then contain information needed to cook the recipe. The software is run on the Gracipe website where the steps is shown as a graphical recipe using a JavaScript frontend.

Because this project was about automatically convert recipes into graphical recipes the input format is not very strict as that would defeat the purpose of it. People should be able to take any recipe in English and convert it into a graphical recipe. This is however difficult to do so some assumptions are made on the format of the input which is described later in this report.

Recipe instructions are usually in imperative mood which means they look like commands. This also means that the first word is very often a verb, an example of such a sentence is “Put the ingredients into a bowl”. This is however not always the case as we can write the same sentence like “In a bowl, put the ingredients” but now the verb is not at the start of the sentence.

Natural language processing is split into 5 different parts, theses parts are

Sentence sementation
: Splits the text into sentences.

Tokenization
: Splits up sentences into words and special characters (e.g. comma, period, colon).

Part-of-Speech tagging
: Gives each word a tag, these tags are different between taggers but are usually different word classes (e.g. nouns, verbs and adjectives).

Entity detection
: Segment and label the entities that might participate in interesting relations with one another.

Relation detection
: Relations between entities are figured out.

More detailed descriptions of each of those steps are given later in this chapter.

The ingredient list and instructions are split up before going through these parts and they go through it individually. There is some difference in the implementation as much of the process is not needed for parsing the ingredient list. This is described in more detail in subsection 2.6.

An input clean-up part was added to the steps to make sure that the software would behave nice. One key aspect of this was the fact that the tagger accepted

only ASCII characters which means that all non-ASCII characters would have to be thrown away. Another function this part serves is when a sentence ends with a temperature such that either “F.” or “C.” are at the end of the sentence it would split the letter and the period and make it into “F .” and “C .” respectively. This was done because the tokenizer was not properly splitting this up into two different tokens when the period comes right after the character.

2.1 Sentence segmentation

Sentence segmentation is fairly simple in this case since each sentence is usually separated by a period. There are exceptions of this but in most of those cases the sentence ends with a newline character. There are two options here for what to do, split the sentences up at newline characters as well or somehow get the user to add periods to the end of each sentence. Both have their pros and cons but they both require the user to make sure the input text is in order. If we were to split the sentence up at a newline character the user has to make sure that the text does not have newline characters in the middle which can happen when copying in text. On the other hand if we were to not split it up at newline characters the user has to make sure that each sentence ends with a period which means that he has to read through the text and add periods.

What was implemented in the end was only splitting at periods and NLTK has a built in function for doing that which is what was used.

2.2 Tokenization

NLTK provides a good tokenizer which does this job for us [1, ch. 3].

There were some complications where it did not correctly split up some tokens. To make up for this the tokens were split up in the clean-up part so that the tokenizer does it correctly.

2.3 Part of Speech Tagging

NLTK provides a part of speech tagger (POS-Tagger) which is generally pretty good but it is very domain specific. For recipe instructions it is not good because of the structure of the sentences.The NLTK POS-Tagger is a supervised learning algorithm which has been taught to classify words based on a database of pre-tagged words. The tagger looks at the word and decides from what it has seen before what the most probable class of that word is. The dataset used to train the tagger is the Penn Treebank tagset.

If for example the input into this is [“Put”, “the”, “chicken”, “into”, “the”, “oven”, “.”] we expect the output to be something like [(“Put”, “VB”), (“the”, “DT”),

("chicken", "NN"), ("into", "IN"), ("the", "DT), ("oven", "NN"), (".", ".")]. The output is a list of tuples where each tuple is a word in the input list and the tag associated to that word.[1] The tags used are the Penn Treebank tags which can be found on their website (`www.ling.upenn.edu`).

We noticed very quickly that the default tagger would not be good for our purposes since the tagger was not tagging verbs correctly, after some research we found out that the tagger is not good at tagging sentences in imperative mood. We saw two solutions to this problem, use a different tagger that is better at tagging sentences in imperative mood or re-tag the words after the initial tagging to compensate for the bad tagging of the default tagger. After some looking around for a different tagger we found a tagger that was better at tagging sentences in imperative mood and we decided to try it out in the hopes of not having to do re-tagging.

This tagger is called the Stanford Tagger and is developed by The Stanford Natural Language Processing Group at Stanford University (`http://nlp.stanford.edu`). To test the accuracy of this we took 100 random sentences from recipes and tagged them by hand and tested the taggers on that data. The accuracy of the Stanford tagger was a lot better than the default tagger which is the reason we switched over to it. More on the test can be found in section 3.1.

We felt that there was still room for improvement. To improve the accuracy we needed to train our own tagger which is exactly what we ended up doing. The data used to train the tagger was only recipe instructions and after training the tagger for a while we used that instead. Most of the work done on the tagger was done by Kai.

Tagging itself does not take a long time generally but to initialize a trained tagger it has to read in a lot of data which can take a bit of time. It is not ideal to initialize a tagger for each input into the software since then we would always have that initialization step which takes a bit of time. A better approach is to have single tagger that serves all processes so that we don't have an initialization step for each process which is exactly what was done. A server was used for tagging sentences and is running on the same machine as the site is hosted on and serves only that machine. This decreased the time taken for converting a recipe by a lot as can be seen in section 3.2.

2.4 Entity detection

Entity detection is all about identifying entities in a sentence. Since we are working with recipes we want to identify the ingredients and the tools used. Some other

[1] In our example the tags are "VB": verb, base form, "DT": determiner, "NN": noun, singular or mass, and "IN": preposition or subordinating conjunction

interesting entities are temperature and time.

This was done in two steps, the first step was to simply go through the tagged text and change the tags of ingredients and tools to “INGRED” and “TOOL” respectively. For this the nouns are compared to a list of ingredients and tools. If the word is among those we know what it should be, otherwise we use a part of NLTK called Wordnet to see if the word is more semantically closer to tool or ingredient.

The next step is what is called chunking which essentially segments and labels multi-token sequences into chunks. NLTK has a regular expression chunker class that can chunk an input sentence according to a certain grammar specified during construction of the chunker. We can then specify what information we want chunked together to make it easier for us to get the information out later.

An important thing to note about chunking and the grammar is that expressions that are defined earlier in the grammar have priority over expressions defined later. A regular expression chunker goes through each expression and tries to match that expression to the sentence and like regular expressions it is greedy and takes as much as it can place in a single chunk and does that. An expression in the grammar can be defined using tags and chunks which means that we can have recursive definitions of an expression.

Because one sentence of instructions can be two or more steps we wanted the chunker to be able to detect when a new step was starting but after a lot of trial and error with the grammar we decided that this was a difficult goal to reach and opted for a much simpler approach where we assume each sentence is a single step. If needed we could then try to find out if there are two or more steps in the sentence in the relation detection step. Since we assume each sentence is a single step the grammar we needed is actually really simple, we just need chunks for ingredients, tools, actions and attributes such as time, heat and temperature. The grammar used can be seen in listing 1

Listing 1: Grammar used for chunking

```
JJ1:          {(<JJ.*|VBG|VBN|VBD>(<CC|,>?<JJ.*|VBG|VBN|VBD>)*)}

AMOUNT:       {<CD>(<TO|CC><CD>)?}

INGREDIENT:   {<DT|AMOUNT>?(<UNIT><IN>?)?<JJ1>*<INGRED>+}
INGREDIENTS:  {<INGREDIENT>(<\.|CC|,|:|IN|TO>+<RB>?<INGREDIENT>)*}
TOOL_PART:    {<IN|TO>?<DT|CD>?<JJ1>?<NN.*|TOOL>+}
TOOLS:        {<TOOL_PART>+}

ACTION:       {<RB.*>*<VB.*>(<CC|,>+<RB.*>*<VB.*>)*(<TO><DT><VB.*>)?<JJ1>?}
              {<RB.*>*<VB.*><PRP><VB.*>}

TIMES:        {<AMOUNT><TIME>}
```

```
            {(<AMOUNT|JJ1>)*<TIME>(<CC><TIMEVB><.*>*<\.|;>|<RBR>)?}
HEATS:      {<ACTION><HEAT><TO><JJ1>}
            {<JJ1>?<HEAT>}
TEMPS:      {<TO|IN><AMOUNT><UNIT>?<TEMP>}

ATTRS:      {(<IN>?<TIMES|TEMPS|HEATS>)+}

STEP:       {<.*>*}
```

This grammar is overly complicated and a few chunks can be skipped. The reason for this is that it was needed earlier when the grammar was a bit more complicated and the entity relation step was made to fit that. If we were to make it simpler we would have to change the entity detection step as well which unfortunately there was no time for.

2.5 Relation detection

Since each sentence was assumed to be a single step in the recipe the relation detection was done in a very simplistic manner. A single step can have a list of ingredients, it can have a tool, it can have an action and it can have attributes. In this part the information is simply gathered from the chunked text, combined and output as a single step. This step is also where we link the ingredient list to the ingredients giving us the amount used (if it was not specified in the text). In this function we have access to the ingredient list extracted before which should have all the ingredients in the recipe. If the ingredient or tool is not included in the ingredient list the Gracipe database is searched.

That was a very high level description of what is done in the relation detection part but there is of course a lot more happening. Often tools are used while not being mentioned in the text, we could have the sentence "Boil the potatoes" which implicitly should be done in a pot. If there is no tool mentioned in the step explicitly the action of the sentence is checked to find out if the tool was given implicitly. The action can also specify an ingredient like in the sentence "Butter the pan" which has to have butter to be able to butter the pan. This was implemented by having a map of certain keywords to tools and ingredients which the software checks.

Another important thing that is implemented in this step to identify sub-steps, that is the step is "Fry cut onions on a pan" we want the onions to be cut with a knife in the output. This is done similar to what we described previously with a map only now we check it for each ingredient.

2.6 Ingredient list

For the ingredient list we don't need to do as much work since ingredients are usually listed in a very specific way. If there is an amount it is specified in the beginning of the line with the appropriate unit, then the name of the ingredient is specified. Sometimes the ingredient list contains a more detailed description of an ingredient, e.g. the ingredient should be cut in some specific way. To find the ingredients in the list of words a semantic similarity check was done on each word to check how closely related it is to being an ingredient. This was done using the Wordnet module in NLTK which allows for semantical distances between words. There was already an ingredient analyzer available on the website which only used database searches to find the unit and ingredient.

2.7 Sample run

For a sample run of the software we imagine that we have the input text given in listing 2.

Listing 2: Example recipe

```
1 tablespoon olive oil
1 teaspoon sugar
425 grams flour
1 teaspoon salt

Put the olive oil, sugar, flour and salt into a bowl.
```

First of all this is split up into two parts, ingredient list and instructions. The ingredient list is then processed to find the ingredients and how much of each ingredient there is. NLTK is used to help find the amount and unit and the Wordnet module in NLTK is used to find what the ingredients are. At this point we have a list of ingredients with the correct amount used in the recipe as well.

The instructions are split into sentences first but since the instructions only have a single instruction this part simply returns a list containing only that sentence. Next up we split the sentence into tokens, the output from this is a list of words and looks like

```
['Put', 'the', 'olive', 'oil', ',', 'sugar', ',',
 'flour', 'and', 'salt', 'into', 'a', 'bowl', '.']
```

The next part is part of speech tagging which takes these these tokens and tags them. The output is a list of tuples where the tuples are the words and their tags. For our example this would be something like

```
[('Put', 'VB'), ('the', 'DT'), ('olive', 'JJ'), ('oil', 'NN'),
```

```
(',’. ',’), ('sugar', 'NN'), (',', ','), ('flour', 'NN'),
('and', 'CC'), ('salt', 'NN'), ('into', 'IN'), ('a', 'DT'),
('bowl', 'NN'), ('.', '.')]
```

The next step would be chunking which with our grammar leads to the chunking shown in the tree in figure 4.

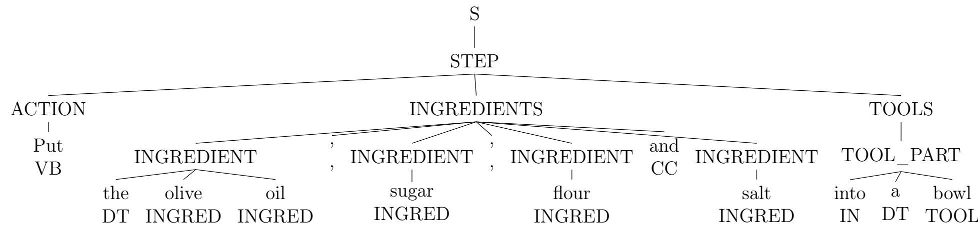

Figure 4: Chunk tree

After we have done this we can check the appropriate chunks for information. We see that the ingredients are “the olive oil”, “sugar”, “flour” and “salt” and by checking the ingredient list we get the correct amount. The tool is “a bowl” and the action is “Put”. The software then returns this as a step and is finished.

3 Results

3.1 Tagger Accuracy

The tagger accuracy was a big problem in the project and tests were done to find out the accuracy of the taggers. A few measures are used to determine the accuracy of the taggers one for the tagger accuracy as a whole and then a couple for the accuracy of the tagger with respect to a certain tag. The accuracy of the tagger as a whole is defined as the percentage of correctly tagged words. For the individual tags accuracy we have precision, recall and F-measure which is the harmonic mean of the precision and recall. Precision is defined as the percentage of words tagged as a certain tag that were correct tagged, or the amount of words correctly tagged as that tag divided by the total amount of words tagged as that tag. Recall is defined as the percentage of words with a certain tag the were tagged correctly.

Most noticeably verbs were being incorrectly tagged which we saw in the recall of the verb tag. The accuracy of the tagger is shown to give an overall performance of the tagger but which tags are responsible for the low accuracy can be seen using the precision and recall.

Results from this can be seen in table 1 where the precision and recall of VB are also shown.

Tagger	Accuracy	VB		NN	
		Precision	Recall	Precision	Recall
Default	81%	0.76	0.16	0.83	0.84
Stanford	96%	1.00	0.66	0.90	1.00
Kai	97%	0.90	0.98	0.98	0.97

Table 1: Accuracy of the part of speech taggers

	Running Time (s)
Server	10.5734
non-server	0.0030

Table 2: Running time of the POS-tagging

We see a huge improvement in the accuracy and this can be attributed to the taggers being better at tagging verbs and nouns accurately. The Stanford tagger actually has a better precision for the verbs and better recall for the nouns then the Kai tagger but overall the Kai tagger is better.

3.2 POS-tagging running time

The part of speech tagging was done using a server and the running time difference between using a server and not can be seen in table 2. The test was done using the same recipe and the time was an average of 20 runs of the tagging on my laptop. Only the time it took to tag the sentence was measured, not the time it took to prepare the sentence for tagging.

The sentence used for testing the speed was the sentence “Put the cookie down.”, a fairly simple sentence that should take no time at all to be tagged. The tagger was in both cases the Stanford part of speech tagger, the non-server tagger was run using the Stanford tagger function built into NLTK. As can be seen in the table the difference in time is substantial which can be traced to having to initialize the tagger every time a sentence needs tagging.

3.3 Accuracy

To determine how well the software was making graphical recipes 20 recipes were converted to graphical recipe form and the amount needed to change was registered. The accuracy was measured using a few criteria, which were the amount of correct ingredients, the amount of correct steps and the amount of correct ingredient amounts. Things that were not taken into account were for example whether

recipe	Ingredients precision	Ingredients recall	amount accuracy	tool accuracy
1	1.00	0.70	0.60	0.66
2	1.00	0.80	0.75	0.66
3	1.00	0.71	0.66	0.66
4	1.00	0.94	0.62	0.50
5	1.00	0.86	1.00	0.75
6	1.00	1.00	0.62	0.71
7	1.00	0.73	0.60	0.50
8	1.00	0.70	1.00	0.83
9	0.95	0.68	0.89	0.56
10	0.67	0.35	0.33	0.55
11	0.94	0.61	0.62	0.55
12	1.00	0.69	0.56	0.89
13	1.00	0.70	0.50	0.50
14	1.00	0.75	0.80	0.50
15	1.00	0.79	0.83	0.86
16	1.00	0.79	0.86	0.83
17	1.00	0.71	0.60	0.67
18	1.00	1.00	0.89	0.71
19	1.00	1.00	0.20	1.00
20	0.94	0.68	0.71	0.73
Average	0.98	0.76	0.68	0.68

Table 3: Experimental results from converting recipes into graphical recipes

the software detected timing, heat and temperature. The results from this can be seen in table 3.

From this we see that the precision for ingredients is really high which means that most of the ingredients that should come actually do come. The recall of the ingredients is however not as high which indicates that ingredients that should not be there come up. The amount and tool accuracy have the same average accuracy which is not very high which indicates that this could be done better.

4 Discussion

The speed at which people can make graphical recipes should increase as a result of this project. The accuracy might not be very high (around 70-80%), but it gives a good starting point for which to create your graphical recipe. Most of the time all the ingredients are already there and the amounts for more than half of

them and the half of the tools are there as well. This is a good template for the graphical recipe which you can change around and play with to get the desired results in the end. Although no tests were done on the actual speed it takes to create a graphical recipe using the converter compared to creating it from scratch we believe this has decreased the time somewhat.

The accuracy of the amounts in the Gracipe was not very high but a part of the reason for this is that if the ingredient is not available in Gracipes database of ingredients the amounts are not registered in the ingredient list. This means that increasing the amount of ingredients in the database would increase this number. Another thing that influences this is the fact that some ingredients are referred to with only part of their name which the software does not always handle correctly. This would be for example if the ingredient list has “chicken breasts” but in the instructions they always refer to it as “chicken”.

5 Conclusion

The software is being tested at the moment by Gracipe testers and will hopefully be released soon. The accuracy of the software can still be increased, especially in finding the tools and ingredient amounts. This can be improved as time goes by and Gracipe already has some plans on how to move forward with this project.

It was an absolute pleasure working with the people at Gracipe, attending their meetings and being a part of the development team of an actual company. It was difficult doing this project alone but thankfully Kai was able to assist me in most parts of the project. We usually ended up splitting up the work and checking over the other persons code afterwards if needed. Kai was often working on creating the web interface and adding features to that while I added the features to the backend. He also trained the tagger himself with only a little help from me at the start of it. If I were to do this again I would prefer doing a project in a small group such that work could be more easily split up.

Natural language processing is a powerful tool in extracting information from written text. This steps of natural language processing are simple but when put together they make a software that can do incredible things like take written recipe texts and turn them into graphical recipes. Making this process reliable is however difficult and can take some time to get right. Especially the chunking and information extraction steps since they define what information is extracted and how. The part of speech tagging has some accuracy errors but most of the time the default tagger is good enough, in our case the sentence structure was such that another tagger was needed.

The software that was implemented shows one way that we can make computers understand what people are writing about and shows the power of natural

language processing. Hopefully this gets people excited about the prospects of communicating with computers using natural language. It's a big task but it can be done partly already using natural language processing.

Bibliography

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.