Assignment 3

Intelligent Word Order Handling for Natural Language Production

Mourad Akandouch
INFOY004
Academic year: 2017-2018
ULB - VUB

Table of contents

UNIVERSITÉ LIBRE DE BRUXELLES ULB VUB VRIJE UNIVERSITEIT BRUSSEL

Section 1

Introduction

Foreword

Two different assignment were proposed for the Natural Language Processing course. The first one consisted in the development of a Grammar for visual question answering while the second one was about handling the ordering of words for natural language production. I have chosen the second one because I like the problems consisting in constraints solving and I was genuinely interested in the second one. Moreover, construct a tool that can predict the best utterances interested me because I could extend the use of that tool in concrete contexts.

The first proposition was also very interesting but when I tried to begin it, I had a lot of trouble to make the needed tools working and the fact that I had to learn a new language while I had tons of other assignments for the other courses definitely led me to choose the second proposition.

★ ★ ★

Introduction

Natural language production's goal is the formulation of a concrete utterance based on semantic representation. The order of words in a sentence is extremely important and different utterances can lead to very different meanings. For instance, *"That grandma ate her healthy food"* is completely different of *"That food ate her young healthy grandma"*. The first sentence is more common and more correct than the second one. Fluid Construction Grammar can provide us correct grammar in order to generate correct sentences. In addition, sometimes, the ordering can be more subtle and need the help of corpus statistics in order to choose what is the best ordering.

For this assignment, we were asked to implement a so-called *render* function that takes a list of constraints as argument and a list of strings representing the words that the function will order. Moreover, the function should be as modular as possible in order to extend the list of constraints it handles. The words that are not hard constrained *(i.e. there exists several possible ordering for those words even when all constraints are satisfied)* will be ordered based on bigram probabilities. The language model for ordering words via probabilities is based on the *Corpus of Contemporary American English.*

Section 2

PART 1 – MODELIZATION AND IMPLEMENTATION

This section will discuss about the implementation of the assignment as well as how I modelled the constraints solving problem. I also give the instruction in order to extend the list of constraints handled.

1. Modelling the problem

I immediately have seen this assignment as a CSP (Constraint Satisfaction Problem) and have used the *python-constraints*[1] library in order to model the constraints and solve them.

I assign to each word (words are the variables of the CSP) a finite domain whose lower bound is zero and the upper bound is the number of words we must order. Hence, at the end of the solving process, we have assigned a number to each word and that number represents its index in the sentence. For instance, let *"I", "cats"* and *"like"* be the words to order. Imagine some constraints whose satisfaction have led to the following solution:

I = 0; cats = 2; like = 1

Then, the solution will be *"I like cats"*.

In order to implements the constraint, we must then create Boolean expression representing the constraint based on the index of the word. For instance, let us consider the *(meets x1 x2)* constraint. The previous constraints forces *x1* to appear right before *x2*. In order to represent that in terms of indices, we write $x1 == x2 - 1$, which means that *x1*'s index must be exactly *x1*'s index minus one.

Now, we see how I have concretely implemented the constraints.

2. The constraints definitions

In order to be as modular as possible, I have defined my constraints in a JSON file that one must place next to the python file. Its name is *constraints-definition.json*. I preferred a JSON file rather than hardcode the list of constraints in the code because doing it that way does not affect the python code.

I define the JSON file as following:

- First, we have a field that enable the creation of "macros" (I will explain that later).
- Then, a field containing a list of joker words used for the bonus functionalities (I will explain that in the last section).
- Finally, another field containing the list of constraints.

For now, the JSON file should be as following:

```
{
  "macros": [],
  "jokers": [],
  "constraints": []
}
```

[1] Documentation can be found here : https://labix.org/python-constraint.

We define one constraint as a dictionary containing at least three mandatory fields. One can add as many field as we want, they will be ignored (Ideal if you want to extend the capabilities of one constraints or create comments).

The mandatory fields of a constraint are:

- **Name** (string): It defines the name of the constraint. For instance, *"precedes", "meets", "between", etc.*
- **#Arguments** (integer): We define here the number of arguments the constraint must handle.
- **Meaning** (array of string): An array containing one or several Boolean expression evaluable by Python's *eval()* function.

Let us consider the *meets* constraint. I have defined it as following (without the last field but it is for the example):

```
{
  "macros": [],
  "jokers": [],
  "constraints": [
    {
      "name": "meets",
      "#arguments": 2,
      "meaning": ["x1 == x2 - 1"],
      "some_ignored_field": "This constraint is cool."
    }
  ]
}
```

We can see that *x1* must be equals to *x2-1* in the meaning field. I have implemented the assignment so that the index of x represents the position of the argument. For instance, when the *render* function will parse the input, if there is a constraint like *(MEETS the-1 cat-1),* then, *x1* will be mapped to *the-1* and *x2* to *cat-1.*

Let us consider one last example. The *(between-2 x1 x2 x3)* constraint, which means that *x1* must be between *x2* and *x3* while the two previous variables must have two words between them. I have defined it as following:

```
{
  "macros": [],
  "jokers": [],
  "constraints": [
    {
      "name": "between-2",
      "#arguments": 3,
      "meaning": ["x1 > x2", "x1 < x3", "abs(x2 - x3) == 3"]
    }
  ]
}
```

Let us focus on the *meaning* field. We can see that in order to represent this constraint, I have used three Boolean expressions. The first one states that *x1*'s index must be greater than *x2's.* The second expression states the opposite with *x3* and the last expression guarantees that there will always be two words between *x2* and *x3.*[2]

[2] For instance, if $x2 = 5$ and $x3 = 8$, then *abs(5-8) is 3*, which gives the possibility for *x1* to take the value 6 or 7.

Finally, recall that the Boolean expressions we write in the *meaning* field must be evaluable by Python's *eval()* function.

Now, let us consider the *"macros"* field that I have defined in the beginning of this section. This field's aim is helping us to use some values from the python's code directly in our JSON file.[3]

For instance, I have defined the constraint *(last x1)* whose meaning is *"the word x1 must be placed at the end of the sentence"*. How to express that if we do not know how many words there will be when we define the constraints' behavior? Well, here comes the utility of the *macro* field. I have created a constant whose value will be mapped to the number of words in the input of the *render* function. Here is how I did it:

```
{
  "macros": [
    {
      "name": "LAST",
      "meaning": "len(self.words) - 1"
    },
  ],
  "jokers": [],
  "constraints": […]
}
```

The downside of the macro field is that we have to know the python code in order to create one. Here, I know that *self.words* is an array where I store the words. Thus, I can map its length to a constant named *"LAST"* and that I can therefore use in my constraints' meaning. For instance, here is how I have defined the *last* constraint.

```
{
  "name": "last",
  "#arguments": 1,
  "meaning": ["x1 == LAST"]
}
```

We can see that I use the constant *"LAST"* directly in the meaning field. During the parsing of the constraints definitions, the code will replace *LAST* by the value returned by *eval("len(self.words) - 1")* every time it sees the word *LAST* in a constraint. More generally, the code will replace all constant declared in the *macros* field by the value returned by *eval(meaning_of_that_constant)* in all the constraints' meaning. We must then be careful and avoid expressions that are not evaluable.

Finally, I have added a constraint directly in the code that ensure that one value cannot be given to two different variables. All variables have different values. The library ensures it easily by adding the following instruction:

My_problem.addConstraint(AllDifferentConstraint())

[3] We can see it as when we define constants in C via the preprocessor.

3. The Language model

In order to create the language model based on COCA corpus, I did it as in the first assignment (but quicker this time thanks to the bigram counts already present). However, I have taken into account the remarks of the first assignment. Now, I save the language model in a file and I do not generate it at each run if the language model can be loaded directly from a file. In addition, in my first assignment, I had used two 2D dictionaries. One representing the word counts and another for the likelihoods. This time, the dictionaries are very huge so I use only one 2D dictionary where I store the pair <count, likelihood> for each bigram.

For instance, if the word "*cat*" follows "*like*" 66 times and its likelihood is 0.003, then we will have an entry like this in my 2D dictionary:

{…, "cat": {…, "like": <66, 0.003>, …}, …}

4. The input

The input of the *render* function is a file containing all the constraints we want to satisfy along with the strings. Note that we cannot have two constraints nor string in the same line. One line per constraint, one line per string.

Here is the content of one of my input files:

```
(STRING that-1 that)
(STRING so-1 so)
(STRING is-1 is)
(STRING romance-1 romance)
(STRING sweet-1 sweet)


(MEETS that-1 romance-1)
(MEETS so-1 sweet-1)
```

We can pass the file to the render function when executing trough command line with:

```
python3 assignment3.py <filename>
```

(Please execute the command line by being in the same directory of the python file. Otherwise, it does not find the JSON file defininig the constraints)

You can also set the default file by modifying the line ~381 of the Python code.

5. The output

The function will produce an output describing the process and will displays the top 10 solutions sorted by their perplexities. If no solution has been found, it will prompt a message warning the user that the constraints could not be satisfied. I display in blue the sentences having a good perplexity, in yellow those having a nearly bad perplexity and in red those having a very bad perplexity.

When all the solutions have a very bad perplexity, the function asks the user if (s)he wants to let the function try to reduce the perplexity by adding some common words in the list of string. This process is repeated until the user write "no" or "n" when the perplexity remains high or when the function have tried all the common words without success.

Section 3

PART 2 – RESULTS, DISCUSSION & BONUS

In this section, I will describe the results I have obtained, how I managed to detect whether the perplexity is bad and what does the *render* function do in order to reduce the perplexity when all the solutions have a very bad perplexity.

1. The common case

First, let us consider a common case where we have good constraints and several words. The input of this case can be found in Appendix 1. The solutions found for that input is the following:

```
>>> 360 solutions found !
>>> Loading language model in order to rank the solutions ... OK !
============================================================
 Printing the most probable solutions based on perplexity:
 * Blue means "Mmmh ... good perplexity."
 * Yellow means "Meh ... Not so good."
 * Red means "Uhh ... I am highly perplex!
============================================================
[ perplexity = 27.04 ]  The girl was stealing his money to give it to her dad
[ perplexity = 28.28 ]  The girl was stealing her money to give it to his dad
[ perplexity = 28.63 ]  The girl was stealing money to give it to her his dad
[ perplexity = 33.34 ]  The girl her to give it to his dad was stealing money
[ perplexity = 34.04 ]  The girl was stealing money her to give it to his dad
[ perplexity = 34.64 ]  The girl to give it to her dad was stealing his money
[ perplexity = 35.19 ]  The girl her was stealing money to give it to his dad
[ perplexity = 36.23 ]  The girl to give it to his dad was stealing her money
[ perplexity = 36.68 ]  The girl to give it to her his dad was stealing money
[ perplexity = 36.69 ]  The girl was stealing money to give it to his her dad
```

We can see that the perplexity is quite low and thus good. I explain later how I detect whether the perplexity is judged good or not.

2. A case without solution

Consider the input described in the Appendix II. With such an input, there clearly is no solution available since we have contradictory constraints.

```
>>> Solving the problem ... OK !
/!\ No solution available. The ordering constraints that you entered could not be satisfied.
```

3. The idioms

I was surprised by the fact that some idioms are detected as having a very bad perplexity. For instance, let us consider the idiom "*A rolling stone gathers no moss*". You can found in Appendix III the input. You can find the result in the next page.

```
>>> Solving the problem ... OK !
>>> 24 solutions found !
>>> Loading language model in order to rank the solutions ... OK !
/!\ The string and ordering constraints that you entered has led to an utterance with a very high perplexity.
>>> Do you want me to try to reduce the perplexity by adding one word in your sentence? (Y/n) n
============================================================
 Printing the most probable solutions based on perplexity:
 * Blue means "Mmmh ... good perplexity."
 * Yellow means "Meh ... Not so good."
 * Red means "Uhh ... I am highly perplex!
============================================================
[ perplexity = 614.8 ]  gathers rolling stone a no moss
[ perplexity = 614.8 ]  rolling stone a no moss gathers
[ perplexity = 618.3 ]  gathers a no moss rolling stone
[ perplexity = 618.5 ]  rolling stone gathers a no moss
[ perplexity = 680.1 ]  no moss gathers a rolling stone
[ perplexity = 680.3 ]  gathers a rolling stone no moss
[ perplexity = 1070. ]  a no moss gathers rolling stone
[ perplexity = 1071. ]  a no moss rolling stone gathers
[ perplexity = 1177. ]  gathers no moss a rolling stone
[ perplexity = 1178. ]  a rolling stone gathers no moss
```

We can see that the worst (in the top 10 solutions) is indeed the idiom. I think that it is normal though. My language model is based on bigrams and having "gathers rolling" may be more probable in common English than "stone gathers". Maybe if I had used 5-grams its perplexity could have lowered a lot since it is a more contextualized in 5-grams.

We can also see that the output asked me if I wanted to try to reduce the perplexity by introducing common words. I answered "n" in order to see the results.

4. How to detect high perplexity (Bonus 1)

In order to detect if some sentence has a high perplexity, I have computed the average perplexity for any sentence of my language model. First, I have generated a sentence containing 65000 words (which is approximately the size of the vocabulary (68784)) and computed its perplexity. It gave me a value around 47, which correspond to the mean perplexity of my language model based on that very long sentence. Then, I consider that all solutions with a perplexity greater than the mean perplexity is yellow (it is not bad but meh.) and the solutions with a perplexity higher than twice the mean value are considered bad.

I think that I could do better than that in order to implement the first bonus[4] but time was lacking. For instance, I think I could have used some statistical notions like the variance or standard deviation in order to be more precise.

5. How to reduce the perplexity (Bonus 2)

In order to reduce the perplexity of the solutions, I ask the user if he wants that the function try to reduce the perplexity by introducing some common words. If the user answers "no", then we print the current solution.

I have searched on Internet the list of the most common word in English and I have copied the top 100. You can see in the JSON file describing the constraints that there is a field whose name is "*jokers*". It contains exactly the top 100 most common word. I call those the "joker" words in the following.

In the python code, I read that list and I try each word of the list. The process is simple:

1. The user wants that the *render* function tries to reduce the perplexity
2. The previous word is removed from the list of words
3. We increment the index of the current joker word in the list
4. I insert the new joker word in the list of words
5. Restart the search process in order to find solutions

[4] *Warn the user if the string and ordering constraints that he entered lead to an utterance with a very high perplexity.*

6. If the best solution still have a very bad perplexity, goto 1, else print the solution.

In addition, I had also tried to relax the system constraints but the assignment stated that we had to insert words (*and it could lead to bad sentences since the constraints are relaxed, even if the perplexity becomes low*), thus I removed that.

Here is an example of an input for which, the solution is bad but after inserting a word, the perplexity becomes low.

In the next figure, we can see that the lowest perplexity is 83 while the perplexity is 440 if I do not introduce the word "of". However, some sentences reduce the perplexity a lot even though the solution does not make always sense.

```
>>> Solving the problem ... OK !
>>> 4 solutions found !
>>> Loading language model in order to rank the solutions ... OK !
/!\ The string and ordering constraints that you entered has led to an utterance with a very high perplexity.
>>> Do you want me to try to reduce the perplexity by adding one word in your sentence? (Y/n) y
>>> I am going to add the word " of " in your sentence ...
>>> Modelization of the problem ... OK !
>>> Solving the problem ... OK !
>>> 20 solutions found !
>>> Loading language model in order to rank the solutions ... OK !
============================================================
 Printing the most probable solutions based on perplexity:
 * Blue means "Mmmh ... good perplexity."
 * Yellow means "Meh ... Not so good."
 * Red means "Uhh ... I am highly perplex!
============================================================
[ perplexity = 83.34 ]  the king of the north
[ perplexity = 167.5 ]  the king north of the
[ perplexity = 200.5 ]  of the king the north
[ perplexity = 252.9 ]  the king the north of
[ perplexity = 329.3 ]  the king of north the
[ perplexity = 454.3 ]  the the king of north
[ perplexity = 454.3 ]  the the king of north
[ perplexity = 465.6 ]  of the king north the
[ perplexity = 642.4 ]  of the the king north
[ perplexity = 642.4 ]  of the the king north
```

I answered "y", so the function tried to insert the word "of" and it gives a better solution.

When I do not want to try to reduce the perplexity, here is what I obtain as solution:

```
>>> 4 solutions found !
>>> Loading language model in order to rank the solutions ... OK !
/!\ The string and ordering constraints that you entered has led to an utterance with a very high perplexity.
>>> Do you want me to try to reduce the perplexity by adding one word in your sentence? (Y/n) n
============================================================
 Printing the most probable solutions based on perplexity:
 * Blue means "Mmmh ... good perplexity."
 * Yellow means "Meh ... Not so good."
 * Red means "Uhh ... I am highly perplex!
============================================================
[ perplexity = 446.7 ]  the king the north
[ perplexity = 1280. ]  the king north the
[ perplexity = 1914. ]  the the king north
[ perplexity = 1914. ]  the the king north
```

6. Coverage and accuracy

Stated from ArXiv, "*Accuracy can be greatly improved by limiting coverage, i.e., by limiting the number of different commands that the system understands. The fewer commands that the system is required to understand (and the less that these commands sound alike), the more likely it is that a well-constructed command will be recognized correctly by the system.*" - https://arxiv.org/html/cs/0006018

Based on that information, I can tell that the accuracy is not very high because I have some sentences that have low perplexity and are ranked higher than those having a higher perplexity but grammatically correct.

Section 4

PART 3 – CONCLUSION

Doing this assignment was funny; I had to imagine how I could make it modular to the point of easily extending the list of constraints. Then, I imagined that JSON file where we can express our constraints and create custom ones. I think that the *render* function could be ameliorated if the language model used consider n-grams with n > 2. In addition, I have tried to create constraints where we can define custom python code in the JSON file as constraint. I succeeded with the python's compile()[5] function but it was a mess to debug and it was impossible to store the bytecode in a variable so I removed it. For the Boolean expression, it is easy because I create lambda expression based those Boolean expressions.

[5] With that function, we can write python code as string and compile it dynamically.

- **Appendix I: The input file for a common case**

```
(STRING the-1 The)
(STRING girl-1 girl)
(STRING was-1 was)
(STRING stealing-1 stealing)
(STRING her-1 her)
(STRING his-1 his)
(STRING to-1 to)
(STRING money-1 money)
(STRING it-1 it)
(STRING to-2 to)
(STRING give-1 give)
(STRING dad-1 dad)

(MEETS the-1 girl-1)
(MEETS was-1 stealing-1)
(PRECEDES stealing-1 money-1)
(MEETS to-1 give-1)
(MEETS give-1 it-1)
(MEETS it-1 to-2)
(FIRST the-1)
```

- **Appendix II : The input file for a contradictory case**

(STRING that-1 that)
(STRING so-1 so)
(STRING is-1 is)
(STRING romance-1 romance)
(STRING sweet-1 sweet)

(MEETS that-1 romance-1)
(MEETS romance-1 that-1)

- **Appendix III : Idiom**

(STRING a-1 a)
(STRING rolling-1 rolling)
(STRING stone-1 stone)
(STRING gathers-1 gathers)
(STRING no-1 no)
(STRING moss-1 moss)

(MEETS rolling-1 stone-1)
(MEETS no-1 moss-1)

- **Appendix IV : Try to reduce**

(STRING the-1 the)
(STRING the-2 the)
(STRING king-1 king)
(STRING north-1 north)

(PRECEDES the-1 king-1)
(BETWEEN king-1 the-1 north-1)