Inf2A Assignment 2

A Natural Language Query System in Python/NLTK

Shay Cohen[1]

Issued 9 November 2018

The deadline for this assignment is **4 pm, Friday 30 November**. Please read through the whole of this handout before you start.

In this assignment, you will use Python(Python2) and NLTK to construct a system that reads simple facts and then answers questions about them. You can think of it as a simple form of both *machine reading* and *question answering*. In the real world, such systems read large amounts of text (e.g. Wikipedia or news sites), populate database with facts learned from that text, and use the database to answer general knowledge questions about the world.[2] We won't require you to build a system that works at that scale! But this assignment should give you a basic idea about how such a system works.

Your completed system will enable dialogues such as the following:

```
$$ John is a duck.

     OK
```

[1]Mostly based on a previous assignment by John Longley, with contributions from Nikolay Bogoychev, Adam Lopez, Toms Bergmanis, and others.

[2]e.g. https://www.google.com/intl/es419/insidesearch/features/search/knowledge.html

```
$$ Mary is a duck.
     OK
$$ John is purple.
     OK
$$ Mary flies.
     OK
$$ John likes Mary.
     OK
$$ Who is a duck?
     John  Mary
$$ Who likes a duck who flies?
     John
$$ Which purple ducks fly?
     None
```

Sentences submitted by the user are either *statements* or *questions*. Statements have a very simple form, but the system uses them to learn what words are in the language and what parts of speech they have. (For example, from the statements above, the system learns that *duck* is a noun, *fly* is an intransitive verb and so on.) Questions can have a much more complex form, but can only use words and names that the system has already learned from the statements it has seen.

In Part A, you will develop the machinery for processing statements. This will include a simple data structure for storing the words encountered (a *lexicon*), and another for storing the content of the statements (a *fact base*). You will also write some code to

extract a verb stem from its 3rd person singular form (e.g. *flies* → *fly*).

Parts B to D develop the machinery for questions. Part B is concerned with part-of-speech tagging of questions, allowing for ambiguity and also taking account of singular and plural forms for nouns and verbs. In Part C you are given a context free grammar for the question language, along with a parser, courtesy of NLTK. Your task is to write some Python code that does *agreement checking* on the resulting parse trees, in order to recognize that e.g. *Which ducks flies?* is ungrammatical. Agreement checking is used in the system to eliminate certain impossible parse trees. In Part D, you will give a *semantics* for questions, in the form of a Python function that translates them into lambda expressions. These lambda expressions are then processed by NLTK to transform them into logical formulae; the answer to the question is then computed by a back-end model checker which is provided for you.

Finally, in Part E, you are invited to supply a short comment on ways in which the resulting system might be improved.

You will need to download the four template files `statements.py`, `pos_tagging.py`, `agreement.py`, `semantics.py` from

`http://www.inf.ed.ac.uk/teaching/courses/inf2a/assignments`

This contains portions of code that are already provided for you, and headers for the pieces of code you have to write. As you proceed, you should check that you understand what the provided code is doing, and what the various NLTK functions are achieving. By the end, you will then have understood all the ideas that have gone into the creation of this system.

Some general points of advice:

- Everything you need to know about Python and NLTK to write your code is covered by the worksheets from earlier lab sessions. Some of the provided pieces of code may feature Python constructs you have not seen before, but you should have little difficulty in understanding them. If you have any questions on the Python/NLTK aspects of this assignment, please ask a demonstrator for help during one of the lab sessions.

- The templates for Parts B,C,D each start with one or more statements that import the material from previous parts (e.g. `from statements import *`). This means that functions defined in one file may be referred to in later files by their plain name. (All your source files should of course be kept in the same directory.)

- You are responsible for *testing* your own code thoroughly before you submit it. Even for experienced programmers, it is rare for any program to work perfectly first time without testing. We have found in the past that students frequently drop marks because of inadequate testing. You are recommended to start a Jupyter Notebook for testing, but remember to auto-reload modules.

- Even if you work mainly on your own machine, you should do a final check that your code works on the DICE machines before you submit it. It is also possible that some parts of the code may not work with different versions of NLTK.

- Part C may be the most difficult part of the assignment. However, even without completing Part C, it is possible to proceed to Part D and obtain a dialogue system that works in simple cases. To do this, simply replace the function `check_node` from Part C by a dummy implementation that always returns `True`.

Part A: Processing statements [25 marks]

In this part you will construct some machinery for processing *statements*. This will involve some simple Python programming to create the necessary data structures, and also the use of regular expression matching facilities in order to build in a little knowledge of English verb forms.

Complete the template file `statements.py` as follows:

1. Create a class `Lexicon` for storing new word stems as we encounter them. The class should provide a method `add(stem,cat)` for adding a word stem with a given part-of-speech category, and method `getAll(cat)` which returns all known word stems of a given category. For instance:

```
lx = Lexicon()
lx.add("John","P")
lx.add("Mary","P")
lx.add("like","T")
lx.getAll("P")           # returns ["John","Mary"]
```

   Your implementation should allow the same word to be stored with more than one POS category (e.g. *fly* might be both a transitive verb and a noun). It should also allow the same `(stem,cat)` pair to be added more than once, although `getAll` should return a list without repetitions (the order of words is not important).

   In use, the parameter `cat` will always be one of five strings denoting part-of-speech categories for word stems:

`"P"`	proper names	(*John, Mary*)
`"N"`	common nouns	(*duck, student*)
`"A"`	adjectives	(*purple, old*)
`"I"`	intransitive verbs	(*fly, swim*)
`"T"`	transitive verbs	(*like, hit*)

(It is intentional that the tagset used here is simpler than that used in Part B onwards. It is also intentional that only open-class words are stored in the lexicon.)

2. Create a class `FactBase` for storing the semantic content of statements. These are represented via *unary facts* such as *duck(John)*, and *binary facts* such as *love(John,Mary)*. You should provide methods

   ```
   addUnary(pred,e1)       addBinary(pred,e1,e2)
   queryUnary(pred,e1)     queryBinary(pred,e1,e2)
   ```

   for adding new facts and for querying whether some fact is among those stored. For instance:

   ```
   fb = FactBase()
   fb.addUnary("duck","John")
   fb.addBinary("love","John","Mary")
   fb.queryUnary("duck","John")              # returns True
   fb.queryBinary("love","Mary","John")      # returns False
   ```

(In actual use, the predicate symbols will be strings such as `"N_duck"` and `"T_love"`, tagged to differentiate predicates arising from different parts of speech, but your implementation should work for arbitrary predicate strings.)

3. We now consider the problem of extracting a verb stem from its 3rd person singular present tense form, henceforth called its 3s form. (E.g. *flies* → *fly*).

   Going in the other direction, below are a selection of rules for deriving the 3s form from the verb stem. These rules have been carefully formulated so that no two verb stems will ever yield the same 3s form, meaning that the process is reversible.[3] For our purposes, a *vowel* is one of the letters *a,e,i,o,u*.

   - If the stem ends in anything except *s,x,y,z,ch,sh* or a vowel, simply add *s* (*eats, tells, shows*).
   - If the stem ends in *y* preceded by a vowel, simply add *s* (*pays, buys*).
   - If the stem ends in *y* preceded by a non-vowel and contains at least three letters, change the *y* to *ies* (*flies, tries, unifies*).
   - If the stem is of the form *Xie* where *X* is a single letter other than a vowel, simply add *s* (*dies, lies, ties* — note that this doesn't account for *unties*).
   - If the stem ends in *o,x,ch,sh,ss* or *zz*, add *es* (*goes, boxes, attaches, washes, dresses, fizzes*).
   - If the stem ends in *se* or *ze* but not in *sse* or *zze*, add *s* (*loses, dazes, lapses, analyses*).

[3] Actually, I haven't been able to think of *any* clear example in English of two different verb stems with the same 3s form. If you find one, let me know!

- If the stem is *have*, its 3s form is *has*.
- If the stem ends in *e* not preceded by *i,o,s,x,z,ch,sh*, just add *s* (*likes, hates, bathes*).

Your task is to turn these rules around so as to retrieve the verb stem from the 3s form. Write a function `verb_stem(s)` that does this, using the `re.match` function for regular expression matching, along with the built-in string processing facilities of Python. If the string `s` cannot be generated by the above rules, `verb_stem(s)` should return the empty string.

Example:

```
verb_stem("flies")       # returns "fly"
verb_stem("flys")        # returns ""
```

4. These rules correctly account for the 3s forms of most common English verbs, although you may be able to think of exceptions. Such rules, however, do not use part of speech information of a given word. For example, according the rules above `verb_stem("cats")` would return `cat`. Your task is to extend the function `verb_stem(s)` to validate that the word accepted as input is indeed used as a verb. To that end you need to use `nltk.corpus.brown.tagged_words()` as the gold standard of English.

   In the Brown corpus, the verb stem and the 3s form are tagged as `VB` and `VBZ`, respectively. Given a possible 3s form as input, if it has never been tagged at `VBZ`, and its hypothesised stem has never been tagged as `VB`, `verb_stem(s)` should

return the empty string. However, if you've seen either the stem form or 3s with the correct tag, return the hypothesised stem. You don't have to check verbs `have`, `are`, or `do`.

To conclude this part, read through the provided code for `add_proper_name` and `process_statement` and check that you understand what it is doing. You will now be able to test your code by processing some simple statements (presented as lists of words) and checking that your lexicon and fact base are updated appropriately.

Part B: POS tagging [20 marks]

We next develop some machinery for identifying all possible parts of speech for a given word with respect to a given lexicon, distinguishing between singular and plural forms. The tags we shall use are as follows:

```
P  A  Ns  Np  Is  Ip  Ts  Tp  BEs  BEp  DOs  DOp  AR  AND  WHO  WHICH  ?
```

The first eight of these are derived from the five POS categories above, distinguishing singular and plural forms for nouns and stems and 3s forms for verbs. (E.g. *cat* Np, *cats* Ns and *like* Tp, *likes* Ts.) The remaining tags are for particular grammatical function words (plus a question mark) as specified by the given list `tagged_function_words`.

Complete the template file `pos_tagging.py` as follows.

1. The main new phenomenon we need to account for here is plural formation of nouns. We shall not attempt anything like a comprehensive treatment of English plurals, but will content ourselves with the following selection of rules:

- For some words the plural form is the same as the singular (e.g. *sheep* or *buffalo*). We call these irregular plural forms `unchanging_plurals`.
- If the noun stem ends in *man*, the plural is obtained by replacing this with *men* (e.g. *woman* → *women*).
- Otherwise, the rules for 3s formation from Part A are applied (e.g. *dogs, countries, ashes*).

First, write a function `unchanging_plurals` which extracts a list of nouns that are unchanging plurals from a corpus provided in `sentences.txt`. Each line of `sentences.txt` contains one sentence. Each word is followed by a delimiter | and its POS tag from Penn treebank tagset[4]:

```
... reporters|NNS are|VBP so|RB busy|JJ ...
```

Note that for this task the only relevant tags are `NN` and `NNS`. You should identify an "unchanging plural" by checking whether a noun appears to be the same both when it is tagged as `NN` and as `NNS`.

Now write a function `noun_stem(s)` which returns the singular form of any plural obtained via these rules. If `s` cannot be generated as a plural by the rules, `noun_stem(s)` should return `""`.

2. Now write a function `tag_word(lx,wd)` which returns a list of all possible taggings for a given word with respect to a given lexicon. For example, for a suitably trained lexicon `lx`, the behaviour should be:

[4]Penn treebank tagset: `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

```
tag_word(lx,"John")      # returns ["P"]
tag_word(lx,"orange")    # returns ["Ns","A"]
tag_word(lx,"fish")      # returns ["Ns","Np","Ip","Tp"]
tag_word(lx,"a")         # returns ["AR"]
tag_word(lx,"zxghqw")    # returns []
```

Your function should use not only the lexicon, but also your functions `verb_stem`, `noun_stem` and the provided list `tagged_function_words`.

Note that your code for Part B should not depend on idiosyncratic features of your own Part A implementation, but only on features specified by the instructions for Part A above. In other words, your code for Part B must work correctly in conjunction with anyone else's (correct) implementation of Part A, not just your own.

Finally, look at the provided code for `tag_words`, a recursive method that provides all possible taggings for a given *list* of words. Experiment with this function. Note that in the worst case, the number of taggings may be exponential in the number of words. Can you find some examples?

Part C: Syntax and agreement checking [25 marks]

The provided context free grammar `grammar` (in `agreement.py`) specifies the language of questions. The terminals of this grammar (enclosed in double quotes) are precisely the POS tags used in Part B. The grammar contains a little redundancy in that e.g. it has both a non-terminal `A` and a terminal `"A"` for adjectives — this is necessary because

NLTK does not allow the right hand side of productions to mix terminals and non-terminals. Note that we have opted to use the colloquial *Who* rather than the more formal *Whom* in questions such as *Who does John like?*

A parser for this grammar is provided by NLTK. The given function `all_parses` returns all possible parse trees for all possible POS taggings of a given word list. A given parse tree `tr` can also be displayed graphically using `tr.draw()`.

Take some time to familiarize yourself with this grammar and experiment with `all_parses`. Notice that multiple parse trees can arise both from POS ambiguity and from structural ambiguity — you may enjoy devising examples.

Your task now is to implement *singular/plural agreement checking* for the resulting parse trees, verifying that verbs agree in number with their subjects. We consider only a single 'number' attribute with the two values `s`,`p`; we use `x`,`y` as attribute variables. Formally, the relevant constraints are captured by the following parameterized version of our grammar rules:

```
S      -> WHO QP[y] QM | WHICH Nom[y] QP[y] QM
QP[x]  -> VP[x] | DO[y] NP[y] T[p]
VP[x]  -> I[x] | T[x] NP | BE[x] A | BE[x] NP[x] | VP[x] AND VP[x]
NP[s]  -> P | AR Nom[s]
NP[p]  -> Nom[p]
Nom[x] -> AN[x] | AN[x] Rel[x]
AN[x]  -> N[x] | A AN[x]
Rel[x] -> WHO VP[x] | NP[y] T[y]
N[s]   -> "Ns"  etc.
```

(In the second rule, we exploit the fact that the stem of a transitive verb coincides with its plural form.)

Complete the template file `agreement.py` as follows.

1. Write a function `N_phrase_num` which returns the number attribute (either `'s'` or `'p'`) for any syntax tree of category N, AN, Nom or NP according to the above rules, assuming the tree satisfies the agreement rules. This can be determined by looking at the number attribute of the *head noun* of the phrase, where proper names are treated as singular. You should write a recursive function that extracts this information from the given syntax tree; two clauses of the definition are provided for you. Note that syntax trees have the NLTK type `Tree`, and the following operations are available for a tree `tr`:

`tr.label()`	the label on the root node of `tr`
`tr[i]`	the subtree with index `i` (starting at 0)
`len(tr)`	the number of subtrees

2. Write a similar recursive function `V_phrase_num` which returns the number attribute of a syntax tree of category T, I, BE, DO, VP, Rel or QP according to the above rules, assuming the tree satisfies the agreement rules. In many cases, this can be obtained as the number attribute of the phrase's *head verb*; in other cases, you may need to think what to do.

   In some cases, the number attribute of a phrase is undetermined by its contents: for instance, the phrase *does John like* (of category QP) could be either singular

or plural, since it can combine with either *Which duck* or *Which ducks*. In such cases, your function should return the result `""`.

Again, two clauses in the definition are provided for you.

3. Now write a function `check_node(tr)` which checks whether the agreement constraints imposed by our parameterized grammar hold for the *top-level* rule application in the tree `tr`. For instance, if the root node is an `S` which expands to `WHICH Nom QP QM`, we just need to check that the number attributes of the `Nom` and `QP` constituents match; this clause is done for you. (Note that the provided function `top_level_rule` obtains the relevant (non-parameterized) production in the form of a string.)

   Your function `check_node` only needs to work for trees of category S, QP, VP, NP, Nom, AN, Rel; it need not work for the preterminals N, I, T etc. Notice that you only need to check constraints arising from the *right hand sides* of the parameterized rules in question — constraints involving the *left* hand side do not need to be checked, since for a rule such as `AN[x] -> N[x]`, the fact that both sides have the same number attribute should follow automatically from the way `N_phrase_num` is computed.

   Once again, two clauses are provided.

The provided function `check_all_nodes` now applies `check_nodes` recursively to check that *all* nodes in the tree are correct according to our agreement rules. These rules can now be used to resolve ambiguities in certain cases: the function `all_valid_parses` filters out any trees that do not satisfy the agreement constraints.

Note that our parsing and agreement checking have both treated the sentence purely as a list of POS tokens rather than actual words. As a preparation for Part D, the provided function `restore_words` reinstates the word stems into the tree, also prefixing them with tags `N_`, `I_`, ...as appropriate. You should experiment briefly with this function in order to understand what it is doing.

Part D: Semantics of the query language [25 marks]

To complete the system, you will need to write a function which converts the trees returned by `restore_words` in suitable lambda-expressions (in the form of strings). These expressions are then passed to NLTK's built-in logic parser and simplifier to obtain logical formulae. Finally, a *model checker* is provided for checking which entities (i.e. which proper names) satisfy the logical formula relative to the fact base, and this provides the answer to the query.

More specifically, your code should translate a complete question into a lambda-expression of type $< e, t >$, which will then be β-reduced down to one of the form `"(\\x.`P`)"`, where P is a formula with free variable `x`. For example:

```
Who is a duck?
  (\x. N_duck(x))
Which orange duck likes a frog?
  (\x. ((A_orange(x) & N_duck(x)) &
        (exists y. (N_frog(y) & T_like(x,y))))
Who does John like?
```

```
(\x. (exists y.((y=John) & T_like(y,x))))
```

One point may seem a little surprising: the semantics will make no use at all of the singular/plural distinction. For instance, the questions *Who is an orange duck?* and *Which ducks are orange?* will have the same meaning: in either case, we are asking for the complete list of orange ducks. Likewise, *Which ducks like a frog?* and *Which ducks like frogs?* are both interpreted as asking for the list of ducks each of whom likes at least one frog. You might be able to think of cases where this policy seems unnatural, but in general it works reasonably and avoids some nasty complications.

The template file `semantics.py` includes the model checker and dialogue session code. Your task is to complete the definition of the recursive function `sem` to provide a compositional translation from trees (as returned by `restore_words`) to lambda expressions. A few sample clauses are provided for you. The clauses that will require the most thought are those involving transitive verbs. Generous use of brackets is recommended!

Phrases of category P should translate to expressions of type e (entities); and phrases of category T to expressions of type $< e, < e, t >>$ (binary predicates) All other phrases should translate to expressions of type $< e, t >$ (unary predicates).

You can test your semantic function by using NLTK to parse and simplify the resulting expressions, as illustrated by the comments in the template file. Once Parts A–D are all completed, you can check that they all fit together by entering a fresh Python session and typing

```
from semantics import *
dialogue()
```

Or just run

```
python2 semantics.py
```

This will launch a dialogue session into which you can type statements and questions at will. Statements and questions end with "." and "?" respectively. Build up your own scenarios and ask complex queries about them, testing whether various kinds of ambiguity are correctly detected and resolved where appropriate.

Part E: Limitations of the system [5 marks]

From your experience of playing with the completed system, you may discover features that you may feel could be confusing or misleading for users. In a plain text file `improvements.txt`, write a short paragraph explaining at least one way in which one might wish to improve the current system without introducing any major new extensions to the language it covers. Also explain whether you think such an improvement would be easy or difficult to implement, given the material you have learned in the course. Justify your answer.

Greater credit will be given for points that are not simply superficial features of the implementation, but engage with the ideas covered by the course.

Submission instructions

To submit your work from the DICE machines, use the commands

```
submit inf2a cw2 statements.py pos_tagging.py
          agreement.py semantics.py improvements.txt
```

Do not submit any other files you write.

An important note: your solution should be implemented so that it runs relatively fast, even with the processing of `sentences.txt`. If it runs in an unreasonable amount of time, you should revise it even if it works correctly. A "reasonable" amount of time has to take into account that the markers run your code manually, and as such, wait until it completes. They have to do it for many students.