LIN 353C: Introduction to Computational Linguistics, Spring 2017, Erk

Homework 3: Regular expressions, part-of-speech tagging, and probabilities

Due: Thursday Feb 23, 2017

This homework comes with a file, `lastname_firstname_hw3.py`. This is a (basically empty) file, called a *stub file*. Please record all the answers in the appropriate places of this file. Please make sure that you save the file in *plain text*, not something like the Word format.

For submission, please rename the file such that it reflects your name. So for example, if your name was "Alan Turing", you should rename it to

```
turing_alan_hw3.py
```

Also do not forget to put your name and EID in the second line of the file.

Please put your Python code into the answers file. You can omit statements that produced an error or that did not form part of the eventual solution, but please include all the Python code that formed part of your solution. **Please submit your homework solution electronically using Canvas.**

If any of these instructions do not make sense to you, please get in touch with the instructor right away.

A perfect solution to this homework will be worth *100* points.

1. **Regular expressions: floating point numbers (10 points)**

General information. Regular expressions describe patterns in character sequences. They exist in many different programming languages. In Python they are implemented in the package *re*, see `https://docs.python.org/3.6/library/re.html`.

And see `http://www.katrinerk.com/courses/python-worksheets/python-worksheet-regular-expression` for a Python worksheet on regular expressions that you can use.

Here are a few things you can do with regular expressions:

- "One out of these characters": For example,

  ```
  [Tt]he
  ```

  will match both "the" and "The". Or

  ```
  [0-9]
  ```

  will match any *single* digit character.

- Matching specific types of character: `\d` matches a single digit, so it does the same as `[0-9]`/ `\w` matches a single letter or digit – it is the same as `[0-9A-Za-z]`. And, quite usefully, `\s` matches a single space (or tab). So for example

  ```
  \d\s\+\s\d\s=\s\d
  ```

  will match an arithmetic expression like "4 + 3 = 7" or "1 + 1 = 2", but not "9 + 10 = 19" (because we specified that the second number would have exactly one digit), and also not "1+1=2" (because we specified that there would be spaces between any number and operator).

  I put a `\` after the "+" above to say "I mean an actual plus, not the regular expression symbol +". We talk about that next.

- The symbol `+` attached to the end of a regular expression says "one or more of these", and `*` means "zero or more of these". So we could improve on our regular expression for arithmetic expressions above by saying

  ```
  \d+\s*\+\s*\d+\s*=\s*\d+
  ```

Now we have said that all the numbers in the expression can have one or more digits, and there can be zero or more spaces between any two items.

The actual problem to solve. For this first problem, give me a regular expression that describes floating point numbers. Please encode the following: A floating point number

- can optionally begin with a "-"
- then come one or more digits
- then, optionally, a period,
- and if there is a period, one or more digits.

Notes: You can use a "verb+?+" after a regular expression to state that it is optional. You can use normal parentheses `( )` to group multiple symbols together in a regular expression. Watch out: period is a special symbol (it matches any character), so to match an actual period, you need to do extra work. Minus is not a special symbol. There are multiple ways to do this task.

For this task, you only need to write down the regular expression. You do not need to encode it in Python. But it may help you to code it in Python so you can try it out. If you would like to do this, you need to import the package:

```
import re
```

Then you can use the function `re.match()` to check whether your regular expression matches a string, in this way:

```
mystring = "ababc"

re.match("[ab]+c", mystring)
```

`re.match()` checks whether the regular expression matches the *beginning* of your string, so this will also succeed:

```
mystring = "ababc"

re.match("[ab]+", mystring)
```

To solve this, you can use `$`, which only matches at the end of the string:

```
mystring = "ababc"

# this will return None, which Python also reads as False:
re.match("[ab]+$", mystring)
```

2. **Regular expressions: onomatopoeia revisited (10 points)**

   In a class worksheet, you worked on the problem of counting the vowels in the word "onomatopoeia". At that point, all we could do was a relatively clumsy solution using `count()`. Now, with regular expressions, you can do better.

   For this problem, please use the Python method `re.findall()` to extract all vowels from onomatopoeia, then count how many matches you got. We want to count the overall number of vowels, so "oeia" needs to contribute 4 counts.

   Here is an example of `re.findall()`:

```
>>> import re
>>> mystring = "12 - 2 + 3 * 4"

>>> re.findall("\d+", mystring)
['12', '2', '3', '4']

>>> re.findall("\d", mystring)
['1', '2', '2', '3', '4']
```

   For this problem, please produce Python code.

3. **Regular expressions and part-of-speech tagging (20 points)**

To solve this problem completely, you will need a working Natural Language Toolkit. If you cannot get the NLTK to work, you can still provide the regular expression patterns and the matching part-of-speech tags; you will just not be able to test how well your tagger does.

General information. Chapter 5 of the NLTK book, which is available at http://www.nltk.org/book/ch05.html , introduces a few simple approaches to part-of-speech tagging. In section 4.2, it introduces the *Regular Expression Tagger*. This tagger encodes knowledge that some prefixes and suffixes are good indicators on the word's part of speech. For example, words ending in "-ion" tend to be nouns, like "invitation", "nutrition", or "ion". The NLTK book lists the following patterns:

```
patterns = [
     (r'.*ing$', 'VBG'),               # gerunds
     (r'.*ed$', 'VBD'),                # simple past
     (r'.*es$', 'VBZ'),                # 3rd singular present
     (r'.*ould$', 'MD'),               # modals
     (r'.*\'s$', 'NN$'),               # possessive nouns
     (r'.*s$', 'NNS'),                 # plural nouns
     (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
     (r'.*', 'NN')                     # nouns (default)
]
```

Read this as: If a word ends in "ing" (that is, it consists of zero or more arbitrary characters, then ing, then end of string), it tends to have the label "VBG".

Note: Something like `(1,2)` is a Python *tuple*. This is pretty much the same thing as a list (except that you cannot append anything to it.) So something like `(r'.*ing$', 'VBG')` is also a tuple of two strings, a regular expression string and a part-of-speech string.

Did you notice the "r" in front of the string? This funny notation tells Python to leave the string "raw", that is, not to interpret it in any way but to pass it on to the regular expression package as is. Python will otherwise interpret patterns involving a backslash, that is, it will

transform "\n" to a "newline". If you have a Windows machine, and you have tried to encode a file path as a string, you may already have had to use "r" in front of the string to keep Python from interpreting the directory name backslashes as special characters.

The actual problem to solve. Please add 10 new regular expressions that link particular character patterns to parts of speech. Add them to the "patterns" list, build a `nltk.RegexpTagger()` as shown in the NLTK book chapter: ¡

```
regexp_tagger = nltk.RegexpTagger(patterns)
```

Test it first on a single sentence, by importing the NLTK version of the Brown corpus, news sentences only, and tagging the 3rd sentence of that collection:

```
from nltk.corpus import brown

brown_sents = brown.sents(categories='news')
regexp_tagger.tag(brown_sents[3])
```

Then, test how accurate it is overall:

```
brown_tagged_sents = brown.tagged_sents(categories='news')
regexp_tagger.evaluate(brown_tagged_sents)
```

(Your score on this problem will not depend on how high your tagger's accuracy is.)

4. **Part-of-speech tagging (30 points)**

   This problem requires text answers. Please just type them in the answer file.

   (a) a. Among the different types of part-of-speech taggers in chapter 5 of the NLTK book was the Default Tagger, which tags all words with the same part of speech. Name *two* reasons why it make sense to have such a tagger.

   (b) The Lookup Tagger in chapter 5 of the NLTK book first uses tagged data to determine the most frequent tag for each of the n most frequent words in the corpus. It then remembers this most frequent tag for each of these words, and always assigns that.

   For less frequent words, it always guesses "noun".

   Chapter 5 of the NLTK book shows a graph (Figure 4.2) that charts the number of most frequent words used against the accuracy that the tagger achieves.

   If you use the 100 most frequent words in the news section of the Brown corpus, then the tagger already reaches an accuracy of 46% on that corpus. Why is it that this simple tagger gets such a relatively high value?

   If you keep adding more and more words with their most frequent tags to this tagger, the model improves up until about 93% accuracy. Why does it not reach 100%?

   And why do you think it is that in the beginning, adding more words makes the model improve fast, but after a while, the performance levels off?

   (c) If you evaluate this Lookup Tagger using all the words in the Brown news section, it achieves an accuracy of 93%. If you evaluate the same tagger (trained on Brown news) on the fiction part of the Brown corpus, you get 80% accuracy. Those are quite different numbers. Which of those two numbers should you trust more? Why?

5. **Accuracy (5 points)**

   The *accuracy* of a part-of-speech tagger is simply the fraction of words that it got right out of all words that it tagged.

   For this problem, you will calculate accuracy by hand. Here are the "true" tags for a sentence:

They	refuse	to	permit	us	to	obtain	the	refuse	permit
PRP	VB	TO	VB	PRP	TO	VB	DT	NN	NN

   Here is what a Lookup Tagger did:

They	refuse	to	permit	us	to	obtain	the	refuse	permit
PRP	VB	IN	NN	PRP	IN	VB	DT	VB	NN

   ("to" is more often a preposition than an infinitival marker, "permit" is more often a noun, and "refuse" more often a verb.)

   Please calculate the accuracy of the tagger on this sentence by hand. Show your work.

   (Incidentally, this example shows another reason why we need part of speech taggers: Some words are pronounced differently based on their part of speech tag, like "object", "refuse", or "permit". If you wanted to build a system that reads a text aloud, having part of speech tags will keep your system from making silly mistakes.)

6. **Bayes' rule (25 points)**

General information. Bayes' rule is a very important rule in statistics and machine learning. It describes how a system (or a person) should revise their opinion after seeing evidence.

Here is a simple example. Say you have been living in Austin, Texas for a while, so you know it does not rain very often here. In Bayesian statistics such insights are expressed in terms of probabilities. So to put some (made-up) numbers to your insight, say that, without having looked out of the window this morning, you would put a probability of 0.9 to the hypothesis "it did not rain last night" (let's call it H_1), and accordingly a probability of 0.1 to the hypothesis "it rained last night" (let's call that H_2). These are your two *hypotheses*. And because we put numbers to the hypotheses before ("prior to") having looked out the window, we call these probabilities the *prior probabilities* of the two hypotheses.

Now say you look out of the window, and you see that the street is very wet. This is your *evidence* E. Bayes' rule then says how you can revise your belief on whether or not it rained last night, using this evidence: You do it by considering *how likely it would be for you to see this evidence under either hypothesis*. Let's first consider the hypothesis H_2, "it rained last night". If H_2 is true, then you would be very likely to see the street be all wet, so the probability of the evidence under H_2 should be pretty high. To put a made-up number ot it, let's say 0.99.

Now, how about H_1? If it did not rain last night, how likely would it be that the street would be very wet? The probability is not zero. Someone's sprinkler may have exploded. Or maybe children have played with some giant water balloons. But the probability is rather low, let's say 0.05.

Bayes' rule now says that to get your *posterior* (after seeing the evidence) belief in a hypothesis, you need to consider both the prior (what you think is the probability of night rain in Austin in general) and the *likelihood*, the probability of the evidence given a hypothesis. We write $P(E|H)$ for "probability of evidence *given* hypothesis", the probability that evidence E will be observed given that some hypothesis H is true:

$P(H|E)$ depends on both $P(H)$ and $P(E|H)$.

Here is Bayes' rule in all its glory, for the case of two hypotheses H_1 and H_2:

$$P(H_1|E) = \frac{P(E|H_1)P(H_1)}{P(E|H_1)P(H_1) + P(E|H_2)P(H_2)}$$

The posterior probability of hypothesis 1, given the evidence, has in its numerator the product of the prior $P(H_1)$ and the likelihood $P(E|H_1)$. In its denominator it has $P(E)$, the probability of the evidence. There we have two cases: Either H_1 is true, then $P(E)$ is $P(E|H_1)$, or H_2 is true, then $P(E)$ is $P(E|H_2)$. If we have two mutually exclusive options (as in this case), then *or* "translates" to the addition of probabilities, which gives us the denominator we have.

So, let's see what we now think about rain last night:

- Numerator for H_1: $P(E|H_1)P(H_1) = 0.05 * 0.9 = 0.05$
- Numerator for H_2: $P(E|H_2)P(H_2) = 0.99 * 0.1 = 0.1$
- Denominator (same for both): sum of the two numerators: $0.05 + 0.1$
- Posterior probability of H_1: $P(H_1|E) = 0.05/(0.05 + 0.1) = 0.33$
- Posterior probability of H_2: $P(H_2|E) = 0.1/(0.05 + 0.1) = 0.66$

So, in this case you think it more likely than not that it did rain last night.

The actual problem to solve. (This is from Ian Hacking's excellent book "An introduction to probability and inductive logic".)

You have been called to jury duty in a town where there are two taxi companies, Green Cabs Ltd. and Blue Taxi Inc. Blue Taxi uses cars painted blue; Green Cabs uses green cars.

Green Cabs dominates the market, with 85% of the taxis on the road.

On a misty winter night a taxi sideswiped another car and drove off. A witness says it was a blue cab.

The witness is tested under conditions like those on the night of the accident, and 80% of the time she correctly reports the color of the cab that is seen. That is, regardless of whether she is shown a blue or a green cab in misty evening light, she gets the color right 80% of the time.

You conclude, on the basis of this information:

- (A) the probability that the sideswiper was blue is 0.8.
- (B) It is more likely that the sideswiper was blue, but the probability is less than 0.8.
- (C) It is just as probable that the sideswiper was green as that it was blue.
- (D) It is more likely than not that the sideswiper was green.

Please *first* choose on of the answers, and record it in your answer file.

Then, do the calculations using Bayes' rule to see what the right answer is. And please show your work in the answer file.

Here is how to use Bayes' rule in this case. You are interested in the probability that the taxi was blue, given that the witness said it was blue. This is a *posterior*. Write it as $P(B|WB)$, with B for "blue" and WB for "witness said blue".

So WB is our *evidence*. We have two *hypotheses* we are weighing: Hypothesis B, it was a blue taxi, and hypothesis G, it was a green taxi.

Your *prior* in this case is just: If a taxi drives by in a minute, how likely is it to be a blue one / a green one? We write these as $P(B)$ and $P(G)$, and use the percentage of taxis in town that are blue vs. green as the prior.

The evidence is "witness said blue", WB. The only piece we are still missing is the *likelihood*, the probability of the evidence given a hypothesis. For hypothesis B, this is $P(WB|B)$, the probability that the witness said "blue" given that the taxi was blue. The experiment that the witness underwent gives us the answer here: This probability is 0.8.

For hypothesis G, the likelihood is $P(WB|G)$, the probability that the witness said "blue" given that the taxi was green. The experiment

that the witness underwent also gives us the answer here: How likely is it that the witness said blue if the taxi was in fact green?

Now plug the numbers into Bayes' rule. You need Bayes' rule for two hypotheses, as in the rain case above.

Does the result match the answer you gave beforehand?