Exam: Introduction to programming (LT2111)

Date: October 19, 2015, 9.00 – 12.00

Course responsible Richard Johansson, Språkbanken, Department of Swedish

Exam accessories None

Grade limits Pass with distinction: 24p, Pass: 15p, Max: 30p

Please note:

- If there is something you don't understand about a question, please ask the course responsible when he comes to the exam room.
- Write legibly (unreadable = wrong)!
- Make sure that your code has a clear indentation (if it is unclear, then the least favorable interpretation will be chosen).
- Number the pages, and start every question on a new page.
- Points will be removed for unnecessarily complicated or unstructured solutions.
- You may use built-in functions and methods, unless otherwise stated, but make sure that you know what they do. If you are unsure, define your own functions.
- If your solution to a question is only partial, please turn it in anyway! Every point counts.

Question 1 of 6: Printing the short words (3 points)

Define a function `print_short_words(lst, n)` that goes through a list `lst` of words, and prints each word whose length is less than `n`. You can assume that every item in the list is a string.

For instance, if we call the function as follows:

```
print_short_words(['Here', 'are', 'some', 'words', '.'], 4)
```

then we get the following output:

```
are
.
```

Question 2 of 6: Problems, problems (5 points)

(a, 2p) We would like a function to compute the total number of course credits a student has accumulated at a university. The course credits are stored in a text file such as this one:

```
Formal_linguistics            7
Programming                   7
Natural_language_processing  15
Dialogue_systems              7
Statistical_methods           7
Information_retrieval         7
```

Our first solution looks like this:

```
def sum_credits(filename):
    total_credits = 0
    with open(filename) as f:
        for line in f:
            course_name, credits = line.split()
            total_credits += credits
    return total_credits
```

Here is how the function should be called:

```
sum_credits('john_smith_credits.txt')
```

If `john_smith_credits.txt` contains the course credits listed above, the result should be 50. However, the code doesn't seem to work as intended. Here is what happens:

```
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Please explain the problem and suggest how to correct it.

(b, 3p) The following code collects the frequencies of the words occurring in a corpus.

```
def get_word_freq_pair(table, word):
    for word_freq in table:
        if word_freq[0] == word:
            return word_freq

def make_frequency_table(words):
    table = []
    for word in words:
        word_freq = get_word_freq_pair(table, word)
        if word_freq:
            word_freq[1] += 1
        else:
            table.append([word, 1])
    return table
```

If we call it like this

```
make_frequency_table(['a', 'rose', 'is', 'a', 'rose', 'is', 'a', 'rose'])
```

then we get the result

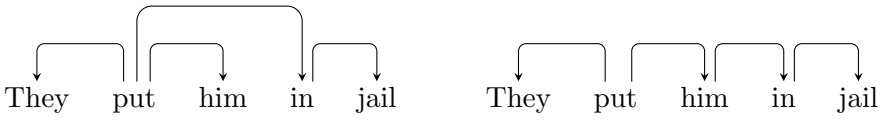
```
[['a', 3], ['rose', 3], ['is', 2]]
```

Strictly speaking this code is correct, but it has a flaw that makes impractical. Please explain what the problem is, and suggest a way to make the program better! (You don't have to write a full reimplementation, just an explanation of what you would do.)

Question 3 of 6: Evaluation of a dependency parser (5 points)

A *dependency parser* carries out a grammatical analysis of a sentence and represents the syntactic structure as a set of links between the words. Each link goes from a *head* word (the grammatically dominant one) to a *dependent* word. In the general case, the links have function labels such as *subject*, *object*, *adverbial*, etc, but in the simplest case, there are no labels. This is the case we consider now.

The following figure shows (a) a correct analysis of the sentence *They put him in jail*, and (b) an incorrect analysis of the same sentence, where an automatic parser has attached the preposition incorrectly.



(a) correct tree (b) incorrect tree

A dependency tree (without function labels) can be represented as a list of integer numbers. Each position in the list corresponds to a token, and the number corresponds to the position of its head. If the number is 0, then the token at that position has no head (that is, it is a root token). For instance, the correct tree (a) corresponds to the list `[2, 0, 2, 2, 4]` and the incorrect tree (b) to `[2, 0, 2, 3, 4]`. In both these lists the first number is 2, because *put* is the head of *They* in both trees.

Your task: Write a function `evaluate_parser(true_tree, predicted_tree)` that computes the *attachment accuracy* of a predicted parse tree with respect to a true tree. The attachment accuracy is defined as the number of correctly attached tokens (that is, their heads are correct) divided by the total number of tokens.

For instance, here is how we would use the function when comparing the bad tree in (b) to the correct tree (a).

```
>>> tree_a = [2, 0, 2, 2, 4]
>>> tree_b = [2, 0, 2, 3, 4]

>>> evaluate_parser(tree_a, tree_b)
0.8
```

The result is 0.8 since 4 tokens out of 5 are correctly attached.

Question 4 of 6: Lexicon-based part-of-speech tagging (5 points)

A *part-of-speech tagger* (PoS tagger) is a program that determines the part-of-speech tag (word class) for a word appearing in a document. One of the simplest imaginable ways that this can be carried out is to use a *tag lexicon* that lists words and their corresponding part-of-speech tags. The lexicon can then be used for automatic tagging simply by looking up words in the lexicon.

(a, 2p) Implement a function `read_tag_lexicon(file_name)` that reads a tag lexicon from a file. Each word in the lexicon, and its corresponding part-of-speech tag, appears on a single line. The function should store the lexicon in some data structure (for instance a list, dictionary, set, or a data structure defined by you) and return this structure.

Here is an example of how the tag lexicon file could look:

```
a        determiner
an       determiner
big      adjective
computer noun
example  noun
file     noun
in       preposition
is       verb
of       preposition
sleeps   verb
this     pronoun
```

(b, 3p) Implement a function `lexicon_tagger(lexicon, file_name)` that goes through the words in a text file and assigns a part-of-speech tag to each word. The file consists of tokenized text, so that each line in the file contains one word. The input `lexicon` is a tag lexicon returned by the function you developed in (a), and you should assign the tag by looking up the word in the lexicon (or `'unknown'` if the word is not listed). The words and their corresponding part-of-speech tags should be printed to the screen.

For instance, assume that we run `lexicon_tagger` with the lexicon listed above and a file containing the following text:

```
this
is
an
example
```

Then the output should look like this:

```
this pronoun
is verb
an determiner
example noun
```

Question 5 of 6: Information retrieval (6 points)

We have collected a collection of web documents and tokenized them. Here is an example of what our collection could look like.

```
collection = [ ['a', 'dachshund', 'is', 'a', 'kind', 'of', 'dog'],
               ['a', 'cat', 'is', 'not', 'a', 'dog'],
               ['Tony', 'is', 'my', 'cat'],
               ['my', 'iguana', 'loves', 'spaghetti'],
               ['a', 'big', 'cat', ',', 'a', 'small', 'dog', ',', 'a', 'black', 'cat'] ]
```

Your task: Write a function `search(collection, query_terms, n)` that goes through the documents in `collection` and checks how many times each document contains a search term from the list `query_terms`. The function should return a list of tuples, where each tuple consists of 1) the number of times any query term occurs in the document, and 2) the document itself. Only the `n` documents where the query terms are most frequent should be returned.

For instance, here is an example of how a pet enthusiast could use the `search` function.

```
>>> search(collection, ['cat', 'dog'], 2)
[(3, ['a', 'big', 'cat', ',', 'a', 'small', 'dog', ',', 'a', 'black', 'cat']),
 (2, ['a', 'cat', 'is', 'not', 'a', 'dog'])]
```

Hint. If you have a list `lst`, you can use the method `lst.count(x)` to determine how many times `x` occurs in `lst`.

Question 6 of 6: Deriving a polarity lexicon from a corpus (6 points)

In *polarity classification*, we want to determine whether the author of a document expresses a positive or negative opinion overall in the document. (This is one of the most basic tasks in *sentiment analysis*; in general, we might be interested in all the different opinions expressed in the document, etc.)

To build a system for polarity classification, one option is to use a *polarity lexicon*, which lists evaluative words and what sentiment they typically express. In our case, we use a polarity lexicon where the polarity values are stored as numbers that express the strength of the sentiment. Words expressing a sentiment with a positive polarity correspond to positive numbers, and vice versa. Here is an example of how such a lexicon could look:

```
polarity_lexicon = { 'good':1, 'bad':-1, 'perfect':2, 'heavenly':4, 'disgusting':-3, ... }
```

(a, 2p) This lexicon can be used in a simple procedure for determining the overall polarity of a document: just sum the polarity values of all the words occurring in the document. If the sum is positive, we say that the overall polarity is positive, and vice versa.
Your task: Write a function `guess_polarity(lexicon, document)` that tries to determine the overall polarity in the document. Its inputs should be a polarity lexicon and a document (a list of words), and it should return either `'positive'` or `'negative'`.

(b, 4p) In practice, it is time-consuming to build a polarity lexicon by hand. A more practical alternative is to derive it from a corpus, and that is what we will do now. Let's assume that we have a corpus of documents, where each document is labeled with the overall polarity it expresses. (Such a corpus can be created e.g. by crawling websites where users can review products.) Here is an example of a corpus containing such polarity–document pairs:

```
labeled_docs = [('positive', ['this', 'movie', 'was', 'good']),
                ('negative', ['this', 'movie', 'was', 'really', 'bad']),
                ('positive', ['it', 'is', 'as', 'good', 'as', 'its', 'predecessor']),
                ('negative', ['it', 'is', 'so', 'bad', 'that', 'I', 'have', 'no', 'words'])]
```

This corpus can be used to build a polarity lexicon automatically using the following procedure:

- start with an empty lexicon (that is, all polarity scores are implicitly zero)
- for every document in the corpus, guess its polarity using the current lexicon and the function you developed in task (a)
- every time you misclassify a positive document, add 1 to the polarity score of each word occurring in the document
- conversely, every time you misclassify a negative document, subtract 1 from the polarity score of each word occurring in the document

Your task: Write a function `build_polarity_lexicon(labeled_docs, n)` that creates a lexicon using the procedure described above. The input `labeled_docs` is a list of polarity–document pairs (as above), and `n` an integer that says how many times you should go through the corpus. The function should return the polarity lexicon.

Congratulations. You just implemented Frank Rosenblatt's famous *perceptron* algorithm.