Analytical Component (30 pts)

Problem 1 (8 pts)

Consider the following co-occurence matrix:

	dog	cat	computer	run	animal	mouse
dog	0	4	0	4	2	2
cat	4	0	0	3	3	10
computer	0	0	0	5	0	5
run	4	3	5	0	3	4
animal	2	3	0	3	0	3
mouse	2	10	5	4	3	0

- a) Using the raw co-occurence counts:
  - Which word is the most similar to '*animal*' using euclidean distance? Show your calculations.
  - Which word is the most similar to '*animal*' using cosine similarity? Show your calculations.

Problem 2 (4 pts) - Homonymy and Polysemy

We discussed the difference between homonyms (words that have multiple unrelated senses) and polysemy (words that have multiple related senses, for example metonymy). Propose an approach that uses WordNet to determine if two senses are related or completely different. You can assume that the two senses are provided as WordNet synsets.

Problem 3 (18 pts): Semantic Composition with Predicate Logic and Lambda Calculus

Consider the sentence "Every dog eats a bone" and the following lexical entries:

[[every]] = $\lambda P.\lambda Q.\forall x\ P(x) \rightarrow Q(x)$ [[dog]] = $\lambda x.dog(x)$ [[a]] = $\lambda R.\lambda S.\ \exists y\ R(y) \wedge S(y)$ [[bone]] = $\lambda y.bone(y)$ [[eats]] = $\lambda T.\lambda x.\ T\ (\lambda z.eats(x,z))$

a) Draw a parse tree for the sentence. Then show how function application can be used to compute a meaning representation for the sentence. Specifically, for each inner node of the tree, show how the meaning representation for this node is composed out of the meaning representations of the children.

Hint: The only tricky part here is the transitive verb *eats.* However,Strictly rewriting the occurences of the lambda parameters with their arguments will yield the correct results.

b) The sentence above has some semantic ambiguity, in the sense that the quantifier scoping is unclear. The meaning representation you just computed for the sentence corresponds to the reading "each dog eats a distinct bone", i.e. many dogs, many bones. The alternative interpration would be that there is only one single bone, that is eaten by every dog. Write this interpretation as a formula in predicate logic (You do not have to show the derivation).

c) In Abstract Meaning Representation (AMR), would there be different graphs for the two different interpretations from b? Explain your answer and draw the AMR graph or graphs (depending on your response).

Programming Component - Lexical Substitution Task (70 pts)

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Please make sure you are developing and running your code using Python 3.

Introduction

In this assignment you will work on a **lexical substitution task**, using both WordNet and pre-trained Word2Vec word embeddings. This task was first proposed as a shared task at SemEval 2007 Task 10 (Links to an external site.)Links to an external site.. In this task, the goal is to find lexical substitutes for individual target words in context. For example, given the following sentence:

"Anyway , my pants are getting *tighter* *every day ."* The goal is to propose an alternative word for **tight**, such that the meaning of the sentence is preserved. Such a substitute could be *constricting*, *small* or *uncomfortable*.

In the sentence

"If your money is *tight* *don't cut corners ."* the substitute *small* would not fit, and instead possible substitutes include *scarce, sparse, constricted*. You will implement a number of basic approaches to this problem. You also have the option to improve your solution and enter your approach into a course-wide competition (see part 6 below). Participating in the competition does not affect grading.

Prequisites: Installing neccessary packages

NLTK

The standard way to access WordNet in Python is now NLTK (Links to an external site.)Links to an external site., the Natural Language Toolkit. NLTK contains a number of useful resources, such as POS taggers and parsers, as well as access to several text corpora and other data sets. In this assignment you will mostly use its WordNet interface. To install NLTK, please follow the setup instructions here (Links to an external site.)Links to an external site.. If you use a package manager (for example, most Linux distributions or macports or homebrew on mac) you might want to install the package for NLTK instead. For example (in macports)

```
$ sudo port search nltk
py-nltk @3.0.4 (python, textproc)
    Natural Language Toolkit

py27-nltk @3.0.4 (python, textproc)
    Natural Language Toolkit

py34-nltk @3.0.4 (python, textproc)
    Natural Language Toolkit

Found 3 ports.

$ sud port install py34-nltk
```

WordNet

Once you have installed NLTK, you need to download the WordNet data. Run a Python intepreter and then

```
$ python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import nltk
>>> nltk.download()
```

This will open up a new window that lets you select add-on packages (data and models) for NLTK. Switch to the corpora tab and select the "wordnet" package. While you are here, also install the English stopword list in the "stopwords" package.

If you have trouble installing the data, please take a look at the documentation here (Links to an external site.)Links to an external site..

Next, test your WordNet installation:

```
>>> from nltk.corpus import wordnet as wn
>>> wn.lemmas('break', pos='n') # Retrieve all lexemes for the noun 'break'
[Lemma('interruption.n.02.break'), Lemma('break.n.02.break'),
Lemma('fault.n.04.break'), Lemma('rupture.n.02.break'), Lemma('respite.n.02.break'),
Lemma('breakage.n.03.break'), Lemma('pause.n.01.break'), Lemma('fracture.n.01.break'),
Lemma('break.n.09.break'), Lemma('break.n.10.break'), Lemma('break.n.11.break'),
Lemma('break.n.12.break'), Lemma('break.n.13.break'), Lemma('break.n.14.break'),
Lemma('open_frame.n.01.break'), Lemma('break.n.16.break')]
>>> l1 = wn.lemmas('break', pos='n')[0]
>>> s1 = l1.synset() # get the synset for the first lexeme
>>> s1
Synset('interruption.n.02')
>>> s1.lemmas() # Get all lexemes in that synset
[Lemma('interruption.n.02.interruption'), Lemma('interruption.n.02.break')]
>>> s1.lemmas()[0].name() # Get the word of the first lexeme
'interruption'
>>> s1.definition()
'some abrupt occurrence that interrupts an ongoing activity'
```

```
>>> s1.examples()
['the telephone is an annoying interruption', 'there was a break in the action when a
player was hurt']
>>> s1.hypernyms()
[Synset('happening.v.01')]
>>> s1.hyponyms()
[Synset('dislocation.n.01'), Synset('eclipse.n.01'), Synset('punctuation.n.01'),
Synset('suspension.n.04')]
>>> l1.count() # Occurence frequency of this sense of 'break' in the SemCor corpus.
3
```

gensim

Gensim is a vector space modeling package for Python. While gensim includes a complete implementation of word2vec (among other approaches), we will use it only to load existing word embeddings. To install gensim, try

```
pip install gensim
```

Or use your package manager. You can find more detailed installation instructions here (Links to an external site.)Links to an external site.. In most cases, installing gensim will automatically install numpy and scipy as dependency. These are numerical and scientific computing packages for Python. If not, you can take a look at the installation instructions https://www.scipy.org/install.html.

pre-trained Word2Vec embeddings

Finally, download the pre-trained word embeddings for this project here GoogleNews-vectors-negative300.bin.gz (Links to an external site.)Links to an external site. (Warning: 1.5GB file). These embeddings were trained using a modified skip-gram architecture on 100B words of Google News text, with a context window of +-5. The word embeddings have 300 dimensions.

You can test your gensim installation by loading the word vectors as follows.

```
>>> import gensim
>>> model = gensim.models.KeyedVectors.load_word2vec_format('./GoogleNews-vectors-
negative300.bin.gz', binary=True)
```

This will take a minute or so. You can then obtain the vector representation for individual words like this:

```
>>> v1 = model.wv['computer']
>>> v1
array([  1.07421875e-01,  -2.01171875e-01,   1.23046875e-01,
         2.11914062e-01,  -9.13085938e-02,   2.16796875e-01,
        -1.31835938e-01,   8.30078125e-02,   2.02148438e-01,
         4.78515625e-02,   3.66210938e-02,  -2.45361328e-02,
         2.39257812e-02,  -1.60156250e-01,  -2.61230469e-02,
         9.71679688e-02,  -6.34765625e-02,   1.84570312e-01,
         1.70898438e-01,  -1.63085938e-01,  -1.09375000e-01,
         ...])
>>> len(v1)
300
```

You can also use gensim to compute the cosine similarity between two word vectors:

```
>>> model.similarity('computer','calculator')
0.333988819665892
>>> model.similarity('computer','toaster')
0.26003765422002423
>>> model.similarity('computer','dog')
0.12194334242197996
>>> model.similarity('computer','run')
0.09933449236470121
```

Alternatively, you can use numpy to compute cosine similarity yourself. Recall that cosine distance is defined as: cos(**u**,**v**) = (**u** · **v**) / (|**u**| |**v**|)

```
>>> import numpy as np
>>> def cos(v1,v2):
...     return np.dot(v1,v2) / (np.linalg.norm(v1)*np.linalg.norm(v2))
>>> cos(model.wv['computer'],model.wv['calculator'])
0.333988819665892
```

Also not necessary for this assignment, you can find some additional information about basic usage of numpy here.

Getting Started

Please download the files and scaffolding needed for the lexical substitution project here hw3_files.zip. The archive contains the following files:

- lexsub_trial.xml - input trial data containing 300 sentences with a single target word each.
- gold.trial - gold annotations for the trial data (substitues for each word suggested by 5 judges).
- lexsub_xml.py - an XML parser that reads lexsub_trial.xml into Python objects.
- lexsub_main.py - this is the main scaffolding code you will complete
- score.pl - the scoring script provided for the SemEval 2007 lexical substitution task.

You will complete the file lexsub_main.py and you should not have to touch any of the other files. You should, however, take a look at lexsub_trial.xml and lexsub_xml.py. The function read_lexsub_xml(*sources) in lexsub_xml.py reads the xml data and returns an iterator over Context objects. Each Context object corresponds to one target token in context. The instance variables of Context are as follows:

- cid - running ID of this instance in the input file (needed to produce the correct output for the scoring script).
- word_form - the form of the target word in the sentence (for example 'tighter').
- lemma - the lemma of the target word (for example 'tight').
- pos - this can be either 'n' for noun, 'v' for verb, 'a', for adjective, or 'r' for adverb.
- left_context - a list of tokens that appear to the left of the target word. For example ['Anyway', ',', 'my', 'pants', 'are', 'getting']
- right_context - a list of tokens that appear to the right of the target word. For example ['every','day','.']

Take a look at the main section of lexsub_xml.py to see how to iterate through the Contexts in an input file. Running the program with a .xml annotation file as its parameter will just print out a representation for each context object.

```
$ python lexsub_xml.py lexsub_trial.xml
<Context_1/bright.a During the siege , George Robertson had appointed Shuja-ul-Mulk ,
who was a *bright* boy only 12 years old and the youngest surviving son of Aman-ul-Mulk
, as the ruler of Chitral .>
<Context_2/bright.a The actual field is not much different than that of a 40mm , only
it is smaller and quite a bit noticeably *brighter* , which is probably the main
benefit .>
...
```

Next, take a look at the file lexsub_main.py. The main section of that file loads the XML file, calls a predictor method on each context, and then print output suitable for the SemEval scoring script. The purpose of the predictor methods is to select an appropriate lexical substitute for the word in context. The method that is currently being called is `smurf_predictor(context)`. This method simply suggests the word *smurf* as a substitute for all target words. You can run lexsub_main.py and redirect the output to a file.

```
$ python lexsub_main.py lexsub_trial.xml  > smurf.predict
$ head smurf.predict # print the first 10 lines of the file
bright.a 1 :: smurf
bright.a 2 :: smurf
bright.a 3 :: smurf
bright.a 4 :: smurf
bright.a 5 :: smurf
bright.a 6 :: smurf
bright.a 7 :: smurf
bright.a 8 :: smurf
bright.a 9 :: smurf
bright.a 10 :: smurf
```

The output indicates that the predictor selected the word smurf for the adjective bright in context 1, etc. You can then run the scoring script (which is written in perl) on the predict file:

```
$ perl score.pl smurf.predict gold.trial
Total = 298, attempted = 298
precision = 0.000, recall = 0.000
Total with mode 206 attempted 206
precision = 0.000, recall = 0.000
```

Unsurprisingly, the smurf predictor does not perform well. Some clarifications:

- The return value of the predictor methods is a single string containing a lemma. The word does not have to be inflected in the same way as the original word form that is substituted.
- The original SemEval task allows multiple predictions and contains an "out of 10" evaluation (accounting for the fact that this task is difficult for human annotators too). For this assignment, we are limiting ourselves to predicting only a single substitute.

Part 1: Candidate Synonyms from WordNet (10 pts)

Write the function get_candidates(lemma, pos) that takes a lemma and part of speech ('a','n','v','r') as parameters and returns a set of possible substitutes. To do this, look up the lemma and part of speech in WordNet and retrieve all synsets that the lemma appears in. Then obtain all lemmas that appear in any of these synsets. For example,

```
>>> get_candidates('slow','a')
{'deadening', 'tiresome', 'sluggish', 'dense', 'tedious', 'irksome', 'boring',
'wearisome', 'obtuse', 'dim', 'dumb', 'dull', 'ho-hum'}
```

Make sure that the output does not contain the input lemma itself. The output can contain multiword expressions such as "turn around". WordNet will represent such lemmas as "turn_around", so you need to replace the _.

Part 2: WordNet Frequency Baseline (10 pts)

Write the function wn_frequency_predictor(context) that takes a context object as input and predicts the possible synonym with the highest total occurence frequency (according to WordNet). Note that you have to sum up the occurence counts for all senses of the word if the word and the target appear together in multiple synsets. You can use the get_candidates method or just duplicate the code for finding candidate synonyms (this is possibly more convenient). Using this simple baseline should give you about 10% precision and recall. Take a look at the output to see what kinds of mistakes the system makes.

Part 3: Simple Lesk Algorithm (15 pts)

Implement the function wn_simple_lesk_predictor(context). This function uses Word Sense Disambiguation (WSD) to select a synset for the target word. It should then return the most frequent synonym from that synset as a substitute. To perform WSD, implement the simple Lesk algorithm. Look at all possible synsets that the target word apperas in. Compute the overlap between the definition of the synset and the context of the target word. You may want to remove stopwords (function words that don't tell you anything about a word's semantics). You can load the list of English stopwords in NLTK like this:

```
stop_words = stopwords.words('english')
```

The main problem with the Lesk algorithm is that the definition and the context do not provide enough text to get any overlap in most cases. You should therefore add the following to the definition:

- All examples for the synset.
- The definition and all examples for all hypernyms of the synset.

Even with these extensions, the Lesk algorithm will often not produce any overlap. If this is the case (or if there is a tie), you should select the most frequent synset (i.e. the Synset with which the target word forms the most frequent lexeme, according to WordNet). Then select the most frequent lexeme from that synset as the result. One sub-task that you need to solve is to tokenize and normalize the definitions and exmaples in WordNet. You could either look up various tokenization methods in NLTK or use the tokenize(s) method provided with the code. In my experiments, the simple lesk algorithm did not outperform the WordNet frequency baseline.

Part 4: Most Similar Synonym (10 pts)

You will now implement approaches based on Word2Vec embeddings. These will be implemented as methods in the class Word2VecSubst. The reason these are methods is that the Word2VecSubst instance can store the word2vec model as an instance variable. The constructor for the class Word2VecSubst already includes code to load the model. You may need to change the value of W2VMODEL_FILENAME to point to the correct file.

Write the method predict_nearest(context) that should first obtain a set of possible synonyms from WordNet (either using the method from part 1 or you can rewrite this code as you see fit), and then return the synonym that is most similar to the target word, according to the Word2Vec embeddings. In my experiments, this approach worked slightly better than the WordNet Frequency baseline and resulted in a precision and recall of about 11%.

Part 5: Context and Word Embeddings (15 pts)

In this part, you will implement the method predict_nearest_with_context(context). One problem of the approach in part 4 is that it does not take the context into account. Like the model in part 2, it ignores word sense. There are many approaches to model context in distributional semantic models. For now, we will do something very simple. First create a single vector for the target word and its context by summing together the vectors for all words in the sentence, obtaining a single sentence vector. Then measure the similarity of the potential synonyms to this sentence vector. This works better if you remove stop-words and limit the context to +-5 words around the target word. In my experiments, this approach resulted in a precison and recall of about 12%.

Part 6: Other ideas? (and competition) (10 pts)

By now you should have realized that the lexical substitution task is far from trivial. In this part, you should implement your own refinements to the approaches proposed above, or even a completely different approach. Any small improvement will do to get credit for part 6. When you submit the project, running lexsub_main.py should run your best predictor for this problem (i.e. you can name that predictor function anything you like). **Shared Task / Competition**: In the spirit of a shared task, we will run a friendly competition to see who can develop the best system. Participation in this competition is optional and does not affect grading. By participating, you agree to share a short explanation of your approach with the class if you win. To participate, do the following: Near the top of your lexsub_main.py file, include a comment in the following format:

```
# Participate in the 4705 lexical substitution competition (optional): YES
# Alias: [please invent some name]
```

The alias should be a short name or identifier that we can use to anonymously release the results of this shared task.