

Programming homework 4: A Lexical Substitution Task

[New Attempt](#)

Due Apr 19 by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip, tgz, and tar.gz **Available** until Apr 23 at 11:59pm

Homework 4 - Lexical Substitution (100 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.**

Unless you have access to your own GPU, we recommended that you complete part of this problem on Google Cloud Platform (GCP). When you run experiments with BERT (and ideally only then, to save your credits), make sure to request a GPU instance to run your code on. You will receive information about your individual GCP voucher by email. We will also distribute setup instructions.

Introduction

In this assignment you will work on a **lexical substitution task**, using, WordNet, pre-trained Word2Vec embeddings, and BERT. This task was first proposed as a shared task at [SemEval 2007 Task 10](http://nlp.cs.swarthmore.edu/semeval/tasks/task10/description.php) [⌵](http://nlp.cs.swarthmore.edu/semeval/tasks/task10/description.php)
[_](http://nlp.cs.swarthmore.edu/semeval/tasks/task10/description.php)
[_](http://nlp.cs.swarthmore.edu/semeval/tasks/task10/summary.shtml)
[_](http://nlp.cs.swarthmore.edu/semeval/tasks/task10/summary.shtml) 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*, *limited*, *constricted*. You will implement a number of basic approaches to this problem and compare their performance.

Necessary packages

[Course Chat](#)

NLTK

The standard way to access WordNet in Python is now [NLTK](#) (Natural Language Toolkit). NLTK contains a number of useful resources as well as access to several text corpora and other data sets. In this tutorial, we will use the WordNet interface. To install NLTK, please follow the setup instructions [here](https://www.nltk.org/install.html) (<https://www.nltk.org/install.html>). On most systems the following

```
$ pip install nltk
```

WordNet

Once you have installed NLTK, you need to download the WordNet data. You can do this by running the following code in a Python shell. Then

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

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

If you have trouble installing the data, please take a look at the documentation [here](#).

<https://www.nltk.org/data.html>. 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')]
```

3/2/23



Ruoxuan Wang 3/2/23

6:07 PM

1



Send

```
>>> l1.count() # Occurrence 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
```

You can find more detailed installation instructions [here \(https://radimrehurek.com/gensim/install.html\)](https://radimrehurek.com/gensim/install.html).

pre-trained Word2Vec embeddings

Download the pre-trained word embeddings for this project here <https://drive.google.com/u/1/uc?id=0B7XkCwpl5KDYNINUTTISS21pQmM&export=download>  <https://drive.google.com/u/1/uc?id=0B7XkCwpl5KDYNINUTTISS21pQmM&export=download> (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.

Alternative link - https://drive.google.com/u/1/uc?id=1iXg_kgfutb9pJwDAEizS8ZT6jcYSpCjy&export=download  https://drive.google.com/u/1/uc?id=1iXg_kgfutb9pJwDAEizS8ZT6jcYSpCjy&export=download (sometimes Google Drive throttles download requests when they are too many in a short amount of time)

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
```

In recent versions of gensim, the `wv` dictionary is no longer available and has been replaced by the `method`

```
v1 = model.get_vector('computer')
```


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: $\cos(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v}) / (|\mathbf{u}| |\mathbf{v}|)$

```
>>> 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
```

Huggingface Transformers


We will use the BERT implementation by Huggingface (an NLP company), or more specifically their slightly more compact model [DistilBERT](https://medium.com/huggingface/distilbert-8cf3380435b5)  [_ \(https://medium.com/huggingface/distilbert-8cf3380435b5\)](https://medium.com/huggingface/distilbert-8cf3380435b5).

We will not train a BERT model ourselves, but we will experiment with the pre-trained masked language model to find substitutes.

```
pip install transformers
```

Part 5 below contains more explicit information about how to use DistilBERT.

Getting Started

Please download the files and scaffolding needed for the lexical substitution project here [hw4_files.zip](https://courseworks2.columbia.edu/courses/174039/files/17258700?wrap=1) [_ \(https://courseworks2.columbia.edu/courses/174039/files/17258700?wrap=1\)](https://courseworks2.columbia.edu/courses/174039/files/17258700?wrap=1) .

[_ \(https://courseworks2.columbia.edu/courses/174039/files/17258700/download?download_frd=1\)](https://courseworks2.columbia.edu/courses/174039/files/17258700/download?download_frd=1) The archive contains the following files:

- lexsu_b_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).
- lexsu_b_xml.py - an XML parser that reads lexsu_b_trial.xml into Python objects.
- lexsu_b_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 (18 pts)

Write the function `get_candidates(lemma, pos)` that takes a lemma and part of speech ('a','n','v','r' for adjective, noun, verb, or adverb) 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', 'obtus
e', '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 remove the `_`.

Part 2: WordNet Frequency Baseline (18 pts)

Write the function `wn_frequency_predictor(context)` that takes a context object as input and predicts the possible synonym with the highest total occurrence frequency (according to WordNet). Note that you have to sum up the occurrence 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 (18 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 appears 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 examples 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 (18 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: Using BERT's masked language model (18 pts)

BERT is trained on a masked language model objective, that is, given some input sentence with some words replaced with a [MASK] symbol, the model tries to predict the best words to fit the masked positions. In this way, BERT can make use of both the left and right context to learn word representations (unlike ELMo and GPT).

We are going to use a pre-trained BERT model. Typically, BERT is used to compute contextualized word embeddings, or a dense sentence representation. In these applications, the masked LM layer, which uses the contextualized embeddings to predict masked words, is removed and only the embeddings are used. However, we are going to use the masked language model as-is.

The basic idea is that we can feed the context sentence into BERT and it will tell us "good" replacements for the target word. The output will be a vector of length $|V|$ for each word, containing a score for each possible output word. There are two possible approaches: We can either leave the target word in the input, or replace it with the [MASK] symbol.

The following example illustrates how to load the pre-trained DistilBERT model. You can find more information in the [API doc.](#) 

(https://huggingface.co/transformers/model_doc/distilbert.html#tfdistilbertformaskedlm)

I recommend that you follow and run the steps of this example to make sure you understand how the masked LM works.

```
>>> import transformers
>>> import tensorflow as tf
>>> model = transformers.TFDistilBertForMaskedLM.from_pretrained('distilbert-base-uncased')
```

Model is actually a full-fledged Keras model, which means we can use it as a plug-in in our own model definition. Here we are going to use the model as-is.

Next, we use the DistilBERT tokenizer to produce the input representation. The tokenizer will split the input into tokens and word parts (as discussed in class). It's important to use this tokenizer because the input to the model must be tokenized in the same way as the training data. The `encode` method tokenizes a string, then adds special symbols [CLS] and [SEP], and then encodes the input into a sequence of integers.

```
>>> tokenizer = transformers.DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
>>> tokenizer.tokenize("If your money is tight, don't cut corners.")
['if', 'your', 'money', 'is', 'tight', ',', 'don', "'", 't', 'cut', 'corners']
>>> input_toks = tokenizer.encode("If your money is tight, don't cut corners")
>>> input_toks
[101, 2065, 2115, 2769, 2003, 4389, 1010, 2123, 1005, 1056, 3013, 8413, 102]
>>> tokenizer.convert_ids_to_tokens(input_toks)
['[CLS]', 'if', 'your', 'money', 'is', 'tight', ',', 'don', "'", 't', 'cut', 'corners', '[SEP]']
```

Note that the target word **tight** is at index 5. Next, we pass the input tokens to the model. Note that the model input has an additional batch dimension and the input should be a numpy ndarray.

```
>>> input_mat = np.array(input_toks).reshape((1,-1)) # get a 1 x len(input_toks) matrix
>>> outputs = model.predict(input_mat)
>>> predictions = outputs[0]
```

The output is a tuple with a single element containing the prediction scores. It is possible to configure the model so that the tuple also contains the hidden states (i.e. the embeddings), but we don't need this functionality.

```
>>> predictions.shape
(1, 13, 30522)
```


So the output is for a single input (first dimension), thirteen tokens (including [CLS] and [SEP], second dimension), and a vocabulary of size 30522. We can look at the predictions for the target word (at position 5).

```
>>> predictions[0,5]
array([-5.256072 , -5.168614 , -5.064958 , ..., -5.1585975, -4.5011725, -2.7322576], dtype=float32)
```

From this, we can get the highest-scoring word indices.

```
>>> best_words = np.argsort(predictions[0][5])[:-1] # Sort in increasing order
>>> best_words
array([ 7376, 13842, 22692, ..., 14821, 25084, 19983])
>>> tokenizer.convert_ids_to_tokens(best_words[:10])
['tight', 'close', 'loose', 'deep', 'big', 'tighter', 'tied', 'firm', 'hard']
```

As expected, the model predicts "tight" as the best match for the input word "tight". However, the other candidates, while similar to "tight" are not really good substitutes in this particular context. Why does this happen? The model has actually seen tight as its input and that information dominates the prediction for the output in that position. This was the reason for introducing the masked LM at all!! We probably could have gotten a similar type of similarity from word2vec. To get contextualized suggestions, we need to mask the target word.

```
>>> input_toks = tokenizer.encode("If your money is [MASK], don't cut corners")
>>> input_mat = np.array(input_toks).reshape((1,-1))
>>> outputs = model.predict(input_ids)
>>> predictions = outputs[0]
>>> best_words = np.argsort(predictions[0][5])[:-1] # Sort in increasing order
>>> tokenizer.convert_ids_to_tokens(best_words[:10])
['stolen', 'wasted', 'worthless', 'dirty', 'yours', 'lost', 'money', 'worth', 'good']
```

These are all words that fit well in this context to make a syntactically correct *and* meaningful sentence. Unfortunately, they don't have much similarity to "tight". By itself, this model won't really help us either. But maybe we can use BERT to rank the WordNet derived candidates! This way, we may be able to combine WordNet's ability to produce a concise set of synonyms (although for all possible senses of the target word), with BERT's ability to take the sentence context into account and select a substitute.

TODO: Take a look at the class `BertPredictor`. When the class is instantiated, it already creates a DistilBERT model and a tokenizer and stores them as instance variables. Write the method `predict(context)` in the class `BertPredictor`, where context is an instance of `Lexsub_xml.Context` that works as follows

- Obtain a set of candidate synonyms (for example, by calling `get_candidates`).
- Convert the information in `context` into a suitable masked input representation for the DistilBERT model (see example above). Make sure you store the index of the masked target word (the position of the [MASK] token).
- Run the DistilBERT model on the input representation.
- Select, from the set of wordnet derived candidate synonyms, the highest-scoring word in the target position (i.e. the position of the masked word). Return this word.

In my experiments, this model performs slightly better than the word2vec approach. The BERT model is doing well at predicting contextually appropriate words, but there are not enough candidates from WordNet limiting performance.

Part 6: Other ideas? (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 (or conceptually interesting approach) 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).