University of Toronto, Department of Computer Science
CSC 485H/2501H: Computational linguistics, Fall 2024

Assignment 2

Due date: 17:00 on Thursday, November 7, 2024.
Late assignments will not be accepted without a valid medical certificate or other documentation of an emergency.
For CSC485 students, this assignment is worth 30% of your final grade.
For CSC2501 students, this assignment is worth 25% of your final grade.

- Read the whole assignment carefully.
- Type the written parts of your submission in no less than 12pt font.
- What you turn in must be your own work. You may not work with anyone else on any of the problems in this assignment. If you need assistance, contact the instructor or TA for the assignment.
- Any clarifications to the problems will be posted on the Discourse forum for the class. You will be responsible for taking into account in your solutions any information that is posted there, or discussed in class, so you should check the page regularly between now and the due date.
- The starter code directory for this assignment is distributed via MarkUs. In this handout, code files we refer to are located in that directory.
- When implementing code, make sure to read the docstrings as some of them provide important instructions, implementation details, or hints.
- Fill in your name, student number, and UTORid on the relevant lines at the top of each file that you submit. (Do not add new lines; just replace the `NAME`, `NUMBER`, and `UTORid` placeholders.)

0. Warming up with WordNet and NLTK (4 marks)

WordNet is a lexical database; like a dictionary, WordNet provides definitions and example usages for different senses of word lemmas. But WordNet does the job of a thesaurus as well: it provides synonyms for the senses, grouping synonymous senses together into a set called a synset.

But wait; there's more! WordNet also provides information about semantic relationships beyond synonymy, such as antonymy, hyperonymy/hyponymy, and meronymy/holonymy. Throughout this assignment, you will be making use of WordNet via the NLTK package, so the first step is to get acquainted with doing so. Consult sections 4.1 and 5 of chapter 2 as well as section 3.1 of chapter 3 of the NLTK book for an introduction along with examples that you will likely find useful for this assignment. You may also find section 3.6 is also useful for its discussion of lemmatization, although you will not be doing any lemmatization for this assignment.

Make certain that you use Python 3 (`python3`). That is where NLTK lives on our machines. You will need to `import nltk`. You may also need to `nltk.download('omw-1.4')`.

(a) (1 mark) A root hyperonym is a synset with no hyperonyms. A synset *s* is said to have depth *d* if there are *d* hyperonym links between *s* and a root hyperonym. Keep in mind that, because synsets can have multiple hyperonyms, they can have multiple paths to root hyperonyms.

Implement the `deepest` function in `q0.py` that finds the synset in WordNet with the largest maximum depth and report both the synset and its depth on each of its paths to a root hyperonym.[1]

(b) (2 marks) Implement the `superdefn` function in `q0.py` that takes a synset `s` and returns a list consisting of all of the tokens in the definitions of `s`, its hyperonyms, and its hyponyms. Use `word_tokenize` as shown in chapter 3 of the NLTK book.

(c) (1 mark) WordNet's `word_tokenize` only tokenizes text; it doesn't filter out any of the tokens. You will be calculating overlaps between sets of strings, so it will be important to remove stop words and any tokens that consist entirely of punctuation symbols.

Implement the `stop_tokenize` function in `q0.py` that takes a string, tokenizes it using `word_tokenize`, removes any tokens that occur in NLTK's list of English stop words (which has already been imported for you), and also removes any tokens that consist entirely of punctuation characters. For a list of punctuation symbols, use Python's `punctuation` characters from the `string` module (this has also already been imported for you). Keep in mind that NLTK's list contains only lower-case tokens, but the input string to `stop_tokenize` may contain upper-case symbols. Maintain the original case in what you return.

[1] Hint: you may find the `wn.all_synsets` and `synset.max_depth` methods helpful.

1. The Lesk algorithm & word2vec (28 marks)

Recall the problem of word sense disambiguation (WSD): given a semantically ambiguous word in context, determine the correct sense. A simple but surprisingly hard-to-beat baseline method for WSD is Most Frequent Sense (MFS): just select the most frequent sense for each ambiguous word, where sense frequencies are provided by some corpus.

(a) (1 mark) Implement the `mfs` function that returns the most frequent sense for a given word in a sentence. Note that `wordnet.synsets()` orders its synsets by decreasing frequency.

As discussed in class, the Lesk algorithm is a venerable method for WSD. The Lesk algorithm variant that we will be using for this assignment selects the sense with the largest largest number of words in common with the ambiguous word's sentence. This version is called the simplified Lesk algorithm.

Algorithm 1: The simplified Lesk algorithm.

```
input : a word to disambiguate and the sentence in which it appears
best_sense ⟵ most_frequent_sense word
best_score ⟵ 0
context ⟵ the bag of words in sentence
for each sense of word do
    signature ⟵ the bag of words in the definition and examples of sense
    score ⟵ Overlap(signature, context)
    if score > best_score then
        best_sense ⟵ sense
        best_score ⟵ score
    end
end
return best_sense
```

Our version represents the signature and context each as bags (also known as multisets) of words. Bags are like sets but allow for repeated elements by assigning each element with a non-negative integer that indicates the number of instances of that element in the bag; this integer is called multiplicity. Because of multiplicity, the bags $A = \{a, a, b\}$, $B = \{a, b, b\}$, $C = \{a, b\}$, and $D = \{a, a, b, b\}$ are all different. (This is not the case for sets, which do not allow multiplicity.) a has multiplicity 2 in A and D but multiplicity 1 in B and C, while b has multiplicity 2 in B and D but multiplicity 1 in A and C.

As with sets, each bag has associated with it a cardinality which is the sum of the multiplicities of its elements; so A and B have cardinality 3 while C has cardinality 2 and D has cardinality 4. The intersection of two bags Y and Z is the bag where, the multiplicity

of any element x is defined as the minimum of the multiplicities of x in Y and in Z. For the bags defined above, C is the intersection of A and B. The union is analogously defined using maximum instead of minimum: the bag D is the union of A and B.

(b) (6 marks) In the `lesk` function, implement the simplified Lesk algorithm as specified in Algorithm 1, including `Overlap`. `Overlap(signature, context)` returns the cardinality of the intersection of the bags `signature` and `context`, i.e., the number of words tokens that the signature and context have in common.

Use your `stop_tokenize` function to tokenize the examples and definitions.

Next, we're going to extend the simplified Lesk algorithm so that the sense signatures are more informative.

(c) (3 marks) In the `lesk_ext` function, implement a version of Algorithm 1 where, in addition to including the words in `sense`'s definition and examples, `signature` also includes the words in the definition and examples of `sense`'s hyponyms, holonyms, and meronyms. Beware that NLTK has separate methods to access member, part, and substance holonyms/meronyms; use all of them.

Use `stop_tokenize` as you did for `lesk`.

(d) (2 mark) This extension should yield improvement in the algorithm's accuracy. Why is this extension helpful? Justify your answer.[2]

Beyond `Overlap`,there are other scores we could use. Recall cosine similarity from the lectures: for vectors $\vec{v}$ and $\vec{w}$ with angle θ between them, the cosine similarity `CosSim` is defined as:

$$\texttt{CosSim}(\vec{v},\vec{w}) = \cos\theta = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|}$$

Cosine similarity can be applied to any two vectors in the same space. In the Lesk algorithm, we compare contexts with sense signatures, both of which are bags of words. If, instead of bags, we produced vectors from the relevant sources (i.e., the words in the sentence for the contexts and the words in the relevant definitions and examples for the sense signatures), we could then use cosine similarity to score the two.

Perhaps the simplest technique for constructing vectors from bags of words is to assign one vector dimension for every word, setting the value for each dimension to the number of occurrences of the associated word in the bag. So $\{a,a,b\}$ might be represented with the vector $\begin{bmatrix}2 & 1\end{bmatrix}$ and $\{a,b\}$ with $\begin{bmatrix}1 & 1\end{bmatrix}$. If we were comparing $\{\texttt{new},\texttt{buffalo},\texttt{york}\}$ and $\{\texttt{buffalo},\texttt{buffalo},\texttt{like}\}$, we might use $\begin{bmatrix}1 & 0 & 1 & 1\end{bmatrix}$ and $\begin{bmatrix}2 & 1 & 0 & 0\end{bmatrix}$, respectively.

(e) (4 marks) In the `lesk_cos` function, implement a variant of your `lesk_ext` function that uses `CosSim` instead of `Overlap`. You will have to modify `signature` and `context` so that they are vector-valued; construct the vectors from the relevant tokens for each in the manner described above.

(Again, use `stop_tokenize` to get the tokens for the signature.)

[2] Hint: consider the likely sizes of the overlaps.

(f) (2 marks) In the `lesk_cos_oneside` function, implement a variant of your `lesk_cos` function that, when constructing the vectors for the signature and context, does not include words that occur only in the signature. For example, if the signature has words {`new`, `buffalo`, `york`} while the context has {`buffalo`, `buffalo`, `like`}, `new` and `york` would not be included; so the signature might be represented with $\begin{bmatrix}1 & 0\end{bmatrix}$ and the context with $\begin{bmatrix}2 & 1\end{bmatrix}$.

(Again, use `stop_tokenize` to get the tokens for the signature.)

(g) (3 marks) Compare how well `lesk_cos_oneside` performs compared to `lesk_cos`. Why do you think this is the case? Justify your answer with examples.

(h) (1 mark) Suppose that, instead of using word counts as values for the vector elements, we instead used binary values, so that {`new`, `buffalo`, `york`} and {`buffalo`, `buffalo`, `like`} would be represented with $\begin{bmatrix}1 & 0 & 1 & 1\end{bmatrix}$ and $\begin{bmatrix}1 & 1 & 0 & 0\end{bmatrix}$, respectively. This is a vector representation of a set.

If we use `CosSim` for such vectors, how would this be related to the set intersection? (You do not need to implement this.)

Finally, let's try to incorporate modern word vectors in place of the bag of words–based method above. Relatively simple models such as the skip-gram model of word2vec can be trained on large amounts of unlabelled data; because of the large size of their training data, they are exposed to many more tokens and contexts. Once trained, word vectors can be extracted from the model and used to represent words for other tasks, usually bestowing substantial increases to performance. They also seem to exhibit some interesting semantic properties; recall the example discussed in class where if we take the vector for king, subtract the vector for man, add the vector for woman, and then ask which existing word vector is closest to the result, the answer will be the vector for queen. It stands to reason that incorporating these vectors might help improve the Lesk algorithm.

(i) (4 marks) In the `lesk_w2v` function, implement a variant of your `lex_cos` function where the vectors for the `signature` and `context` are constructed by taking the mean of the word2vec vectors for the words in the signature and sentence, respectively. Count each word once only; i.e., treat the signature and context as sets rather than multisets.

(Again, use your `stop_tokenize` to get the tokens for the signature.)

You can run your implementations on the evaluation set that we provide by running `python3 q1.py`. This will skip any unimplemented functions and display scores for the implemented ones. Report your scores in your written submission.

(j) (2 marks) Alter your code so that all tokens are lowercased before they are used for any of the comparisons, vector lookups, etc. How does this alter the different methods' performance? Why?

(Do not submit this lowercased version.)

2. Word sense disambiguation with BERT (22 marks)

word2vec associates a vector with each word type. Newer models, such as ELMo, GPT, and BERT instead produce vectors for each word token. Another way of saying this is that vector representations produced by the latter models are conditioned on context and will differ depending on the context in which the word appears. Consider the two sentences Let's play a game and Let's see a play. word2vec will produce the same vector for play in both cases even though they have different senses. This is why you had to average all of the relevant vectors in the signature or sentence for `lesk_w2v` rather than, for example, relying only on the vector for play. Clearly, there's a parallel to how ambiguous words can have their senses resolved only in context.

(a) (4 marks) Is context really necessary? Assuming all that is available are wordforms and lemmata—no dependencies, semantic roles, parts of speech, etc.—can you give an example of a sentence where word order–invariant methods such as those you implemented for Q1 will never be able to completely disambiguate? If so, what is the more general pattern, and why is it impossible for the above methods to provide the correct sense for each ambiguous word? Be clear about your reasoning.[3]

But BERT (as is the case with other contextual models) doesn't produce the same vector for every instance of a particular word sense; the same vector will be produced only in the exact same context. So we cannot simply glance at the vector produced for an ambiguous word in order to determine its sense. We might consider using BERT vectors in a similar manner as we did the word2vec vectors for `lesk_w2v` above, taking the mean of the vectors to produce a vector representation for a word sequence; this turns out to perform worse than any of the methods above.[4]

Instead, suppose that we had a vector associated with every sense. If the sense vectors are related to corresponding word token vectors in some way, we could then compute similarities between possible sense vectors for a particular word token and its corresponding contextual word vector and then select the sense with the highest similarity. A simple method that works well is to run the model (BERT, in our case) over each of the sentences in a sense-annotated training set. We can gather the word tokens associated with all occurrences of a sense and take their average to represent that sense.

(b) (10 marks) Implement `gather_sense_vectors` in `q2.py` to assign sense vectors as described above.

(c) (2 marks) In the docstring for `gather_sense_vectors`, we point out that sorting the corpus by length before batching is much faster than leaving it as-is. Explain why this is the case.[5]

[3] Hint: re-read the preceding paragraph with this question in mind.
[4] Why?
[5] Hint: think about padding.

(d) (4 marks) Implement `bert_1nn` in `q2.py` to predict the sense for a word in a sentence given sense vectors produced by `gather_sense_vectors`. Keep in mind the note in the docstring about loop usage.

For this assignment, you will be using the BERT model, though the direct calls to it have been implemented for you in given helper functions. Pay attention to the notes, etc. in `q2.py`, as they will help with and fully specify proper usage.

Finally, beware that the version of word sense disambiguation in this assignment is restricted compared to what would have to be done for the general case of disambiguating ambiguous words in arbitrary sentences. In particular, the code you've written assumes that it is known which words need to be disambiguated; the selection of ambiguous words in the sentence has already been done for you.

(e) (2 marks) Think of at least one other issue that would come up when attempting to use the code for this assignment to disambiguate arbitrary sentences. You may consider either the Lesk variants from Q1 or the BERT-based method here (or both).

3. Understanding transformers through causal tracing (16 marks)

In this question, you will deepen your understanding of how large language models (LLMs) generate responses by examining the model's decision-making process. For example, when asked, "The Eiffel Tower is located in the city of ____," how does the model decide that the answer should be "Paris"? You will analyze the roles of different components, such as attention mechanisms and MLPs, and determine how different layers contribute to the model's overall functionality. Specifically, you will employ the causal tracing method, as discussed in class, to investigate how information flows within the model to produce its outputs.

An autoregressive Transformer model over the vocabulary V, takes a sequence $x = x_1, ..., x_T$, $x_i \in V$ as input and returns a probability distribution of the next token in the sequence. Within the Transformer, the i^{th} token at l^{th} layer is represented by the hidden state vector h_i^l. The model's final output $y = decode(h_T^l)$ is obtained from the hidden state corresponding to the last token. The dependencies between the hidden states can be visualized by causal graph (Figure 1). In this example, GPT-2 can predict the correct answer "`Paris`" by giving the prompt "`Eiffel Tower is located in the city of`".

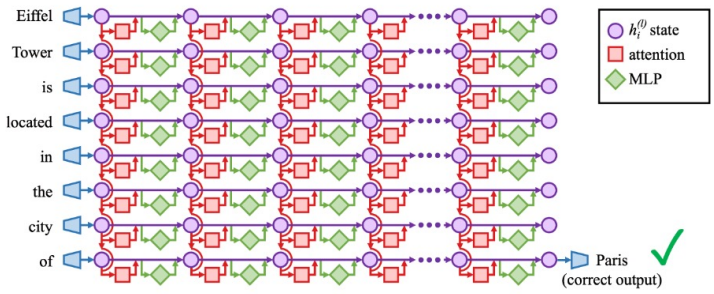


Figure 1: Causal graph.

Causal tracing is a technique used to analyse and understand how information flows through a language model. This process involves three main steps:

- Clean Run: A factual prompt x is passed into the autoregressive language model as is, without modification. This provides a baseline run, as demonstrated in Figure 1.

- Corrupted Run: In this step, the source tokens of the prompt s are obfuscated immediately after the embedding layer h_i^0. This is done by adding random noise $h_i^0 = h_i^0 + \varepsilon$ to each token, corrupting all hidden states. Running the language model in this corrupted state typically leads it to provide an incorrect answer. The corrupted hidden states are denoted as h_i^l*.

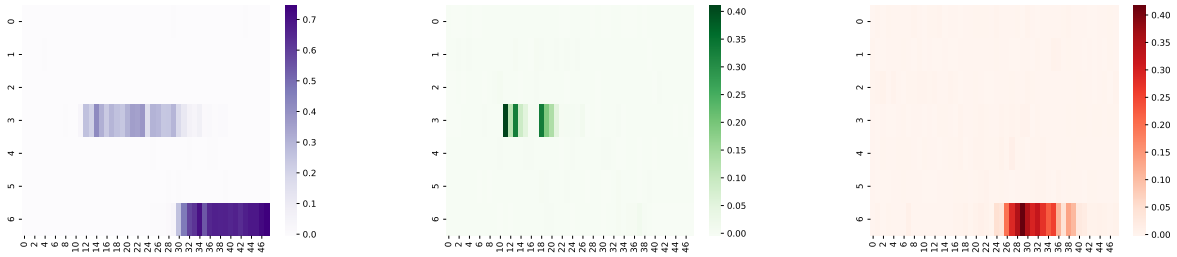
- Corrupted with Restoration Run: Here, we run the language model with corrupted inputs, but we "restore" the hidden state at a chosen token position $\hat{i}$ and layer $\hat{l}$ to

its clean state. This allows us to observe how restoring specific hidden states can aid in recovering the correct output, even when many other states remain corrupted.

Under all three settings above, the language model will return the probability of the last token. Let us use $\mathcal{P}, \mathcal{P}*, \mathcal{P}_{*,clean_i^l}$ to denote the distribution probability. The indirect effect(IE) of a hidden state is the difference between the probability of the last token when the state is corrupted and when it is switched to its clean version. IE is represented as: $IE = \mathcal{P}_{*,clean_i^l} - \mathcal{P}*$.

(a) (3 marks) Implement `get_forward_hooks`.

(b) (5 marks) Implement `causal_trace_analysis` to compute the impact of states, MLP and attention.



(a) State. (b) MLP. (c) Attn.

Figure 2: Causal tracing result for the prompt "The Space Needle is located in" and the output "Seattle".

We can use a tuple to represent knowledge, $t = (s, r, t)$ where s is the source, r is the relation, and t is the target that the language model needs to predict. Figure 2 shows the example output if using GPT2-xl with the prompt "The Space Needle is located in" and the output "Seattle".

(c) (1 mark) Report your generated causal tracing result plots for the prompt "The Eiffel Tower is located in the city of" with the output "Paris" in your report.

(d) (3 marks) Experiment with different sizes of GPT-2 models (e.g., GPT-2 small, medium, large, and XL) to examine how model size impacts causal tracing patterns. In your report, address the following:

- At what model size do you observe that the causal tracing pattern no longer appears?
- Based on your observations, discuss potential reasons for how and why this change in causal tracing patterns occurs as the model size increases or decreases.

(e) (2 marks) Using GPT-2 XL, experiment with various prompts to identify prompt types that result in a causal tracing pattern similar to the one illustrated in Figure 2. Document your findings with examples and discuss what characteristics of the prompts might contribute to this similarity.

(f) (2 marks) Similarly, for GPT-2 XL, explore different prompts and tasks to find cases where the causal tracing pattern is absent or significantly diminished. For these cases, describe the prompt/task and hypothesize why the pattern does not emerge.

Discuss any trends or patterns you identified, and reflect on the broader implications of how language models process, store and generate factual information obtained from pretraining.

Notes

- Running the code for Q1 should not take too long, but if you would like to iterate faster, caching tokens for signatures and contexts can be an effective speedup.
- The BERT code for Q2 can be a little slow and memory-intensive, so it's recommended that you run it on a GPU. (That said, it should run on CPU in less than an hour or so, so it's definitely viable to do so if needed or preferred.) The code we've provided for this assignment will automatically use the GPU if it's available; you should never have to specify the device of a `Tensor` or call `.clone().detach()` on one. Similar to A1, you can run your Q2 code on a GPU-equipped machine via ssh to `teach.cs.toronto.edu`; use the `gpu-run-q2.sh` script to do so. As a reminder, the GPU machines are shared resources, so there are limits on how long your code is allowed to run.
- Do not alter the function signatures (i.e., the names of the functions or anything about the parameters that they take), and do make sure that your implementations of the functions return what the functions are specified to return (i.e., no extra return objects). Also, make sure your submitted files are importable in Python (i.e., make sure that `` `import q1' `` doesn't yield errors, and likewise for the other questions/files).
- Do not add any extra package or module imports. If you really would like to use a certain package or module, ask on the Discourse forum with a brief justification.
- You may add any helper functions, global variables, etc. that you wish to the files for each question (`q0.py`, `q1.py`, etc.), but do not allow your code to become overly convoluted and make sure it is still readable so that we can trace and understand your logic. You will not be submitting the other Python files, but avoid making changes to them, as your code's behaviour will be evaluated against those files as they are in the starter code.

What to submit

This assignment is submitted electronically via MarkUs. You should submit a total of five ("5") required files as follows: The files you are to submit are as follows:

- `a2written.pdf`: a PDF document containing your answers to questions 0a, 1d, 1f, 1h, 2a, and 2d.

  This PDF must also include a typed copy of the Student Conduct declaration as on the last page of this assignment handout. Type the declaration as it is and sign it by typing your name.

- `q0.py`: the (entire) `q0.py` file with your implementations filled in.
- `q1.py`: the (entire) `q1.py` file with your implementations filled in. Again, do not include the alterations that you implement for question 1h.
- `q2.py`: the (entire) `q2.py` file with your implementations filled in.
- `q3.py`: the (entire) `q3.py` file with your implementations filled in.