NLP course Assignment 2: Contextualized Vectors, Parts of Speech, and Named Entities

Daniel Bazar 314708181

Lior Krengel 315850594

0) Warmup

1) We encoded the sentence “I am so <mask>” and:
   1.1) We extracted the vectors for “am” and for “<mask>”. Both vectors are partially represented because of their shape (768):
   am:
   [2.9096e-01, 9.2609e-02, 1.4434e-01, -1.8008e-01, 5.1247e-01,
   ...
   1.3153e-01, -8.0886e-02, 3.9851e-02]
   <mask>:
   [3.4503e-01, -1.1836e-01, -1.9594e-02, -8.2120e-02, 7.9033e-01,
   ...
   2.3184e-01, -3.3112e-02, 2.8167e-02]
   1.2) We extracted the top-5-word predictions for “am” and for “<mask>” and their probabilities:

   am:

am	0.9999
is	3.9379e-05
'm	2.9937e-05
was	8.6890e-06
feel	8.5508e-06

   <mask>:

sorry	0.3084
proud	0.0649
grateful	0.0581
happy	0.0448
blessed	0.0324

2) We find two sentences that share the same word, such that the cosine similarity between the word vectors in the two sentences is **very high**:
   Sentence1: 'I love you' Sentence2: 'I love him' similarity: 0.9897
3) We find two sentences that share the same word, such that the cosine similarity between the word vectors in the two sentences is **very low** (low is relative):
   Sentence1: 'The fission of the cell could be inhibited with certain chemicals.'
   Sentence2: 'His cell phone worked, so he spoke with his parents and sister-in-law.'
   similarity: 0.8418
4) We find a sentence with n words, that is tokenized into m > n tokens by the tokenizer:
   original sentence: Didn't I tell you it's gonna be a rock 'n' roll weekend with lots o' fun, and we'll gather 'round the campfire, singin' our favorite songs 'til the break o' dawn? **(n=31)**
   tokenized sentence: ['<s>', 'Did', 'n', "'t", ' I', ' tell', ' you', ' it', "'s", ' gonna', ' be', ' a', ' rock', " '", 'n', "'", ' roll', ' weekend', ' with', ' lots', ' o', "'", ' fun', ',', ' and', ' we', "'ll", ' gather', " '", 'round', ' the', ' camp', 'fire', ',', ' sing', 'in', "'", ' our', ' favorite', ' songs', " '", 'til', ' the', ' break', ' o', "'", ' dawn', '?', '</s>'] **(m=49)**

1) Part-of-speech tagging

In this part of the assignment, we will explore the notion of part-of-speech tagging.
The “catch” in this assignment is that we don’t do it in the standard way.
Instead of train a classifier to predict the correct part-of-speech tag from vector representation, in this assignment we will experiment with predicting parts of speech of words without training any classifiers.
The general approach we chose to deal with the problem is to maintain a dictionary whose keys are ’context’ (words, bigrams, previous pos, and so on) and their values are their POS distribution in the train data. In inference, we tagged each word based on this dictionary. We tackled the problem of unknown words that are out of our dictionary, by taking the most common POS tag based on the context.
This is also the main difference between the taggers in each task in this section, how we handled the unknown words.

As will be described, we experimented with different methods and techniques to find the best approach based on the accuracy of development data.

1.1) No word vectors

In this section, we are not allowed to use any word vectors at all.
First, we created a dictionary with every word and its corresponding POS. To determine the tag we tried two methods, the most common (argmax) and sampling. We did it both to known and unknown words and as we can see below, argmax is better:

		Know words	
		sampling	argmax
Unknown words	sampling	0.884	0.904
	argmax	0.89	**0.912**

Accuracy of our predictions by ‘decoding’ method

In addition, we checked whether lowercasing the data improves results. We assumed that it can generalize better because it “contains” words in various cases and not just the certain cases seen in training. However, the result shows the opposite. The accuracy of this method was just **0.858**.
This showed that the case of words plays a crucial role in POS tagging, as it probably holds additional information. For example, distinguish proper nouns from common nouns, convey semantic differences and more. Also, in case of unknown words, we tried to predict the POS based on another case of the word if it exists in our data, but it also hurt performance.

Furthermore, in attempt to improve prediction, we use the same architecture, but this time the ‘context’ is also the previous POS, a bigram model. This approach improved prediction as we get accuracy of **0.929**.

As all above in mind, we know that a bigram model, with “argmax decoding” and no lowercasing is our best settings, we continue with this to the next task.

1.2) Static word vectors

As before, but now we are allowed to use static vectors. With this, we can handle unknown words a little differently. Instead of predicting their POS based on the general distribution, we can use static word embedding algorithms to retrieve the most similar words to them that we know, and then assign the tag based on them. We experimented with different top-K neighbors and chose based on the mode of the neighbors predictions. We explored k from 1 to 9.

For choosing the word embedding model we took all the models in the *gensim* package, and we checked their size and how much their overlapped with our training data. We assumed that bigger vocabulary, with smaller overlapping with train data, will generalize more as it will probably contain more unknown words. We chose this way to choose the model and not by evaluating on dev data (although we did that too) because we were afraid maybe it would fit too much to dev and will generalize less to unseen data. After filtering nonrelevant models (like Russian and more) our chosen model was: *"glove-wiki-gigaword-300"*.

Recall that we use the bigram architecture, with argmax when word and previous POS is known.

Top-k	Accuracy
1	0.9325
2	0.9325
3	**0.9328**
4	0.9329
5	0.9329
6	0.9330
7	0.9332
8	0.9332
9	0.9334

All k's give similar results. Bigger k means longer running time so with this tradeoff we chose k=3.

1.3) Contextualized word vectors

As before, but now we are allowed to use the output of a roberta-base model.

With this, we can use the same technique as above (1.2) but this time the word embedding for unknown word, is contextualized to the sentence, hence, it can handle disambiguation of words and as a result improve performance.

We used the same architecture as above and explored k from 1 to 9.

Top-k	Accuracy
1	0.9363
2	0.9363
3	0.9373
4	0.9377
5	**0.9381**
6	0.9379
7	0.9383
8	0.9386
9	0.9385

All k's give similar results. Bigger k means longer running time so with this tradeoff we chose k=5.

To sum up, a model that uses a Contextualized word vectors is our best POS model.

2) Named Entities Recognition

With our previous experiment with POS tagging in mind, we began with the naïve strategy. We built a word counter of the NER tag distribution and took the argmax. In first place, we decided not to predict spans but rather tag each word and if it's a NER tag different than 'O' so it's inside the span. This approach gave us our baseline performance for the rest of this section. With this approach we got the following results:

Overall Accuracy (including 'O'): 93.88%

	Precision	Recall	F1
All-types*	**71.66%**	**65.95%**	**68.69%**
MISC	73.40%	69.74%	71.52%
PER	68.55%	53.96%	60.39%
LOC	83.24%	78.93%	81.03%
ORG	59.39%	62.04%	60.69%

*excluding 'O'

Now, if we encountered missing word, we first Tried to inflect them (capitalize or lower) to a form that is in our train data. We achieved slightly improved results. (To be precise, it improved recall and worsened precision. It's make sense because we tag more words, so we "cover" more but at a price of mistakes):

Overall Accuracy (including 'O'): 93.94%

	Precision	Recall	F1
All-types*	**71.24%**	**67.06%**	**69.09%**
MISC	72.99%	69.74%	71.33%
PER	68.66%	54.83%	60.97%
LOC	82.99%	81.00%	81.98%
ORG	58.25%	62.94%	60.50%

*excluding 'O'

Now, like in the POS tag we look for a method that includes word vectors and some context. We learned in the previous part that the best approach was using contextualized word vectors for tagging the out of vocab words and a bigram model that depend also on the previous POS. But before doing that, we tried some new ideas. We thought about encoding the whole data, doing dimensionality reduction, and then clustering, in an attempt to create more features. but we encountered a problem, encoding every word in the train data is very heavy in terms of running time, and its before we applied the dimensionality reduction and the clustering. and all of that needs to be computed also in inference time. So, after some effort, we decided to leave it. Another concept we tried is to expand our dictionary. We analyzed our prediction mistakes and found out that a lot of it was places and names we didn't see in our train data. We can't use other data sources, so the approach we adopted was to "inflate" our data by obtaining the most similar words to our most frequent words (that have a NER tag different than 'O') that didn't appear in our train data. We use the contextualized model to retrieve those words. We experimented with different parameters of the top-k neighbors and the most frequents words and at the end we expanded our data with the top-5 words most similar to the 1000 most frequents word in our data, meaning expanding it by ~5000 words of different names, locations, organizations and so on. But surprisingly, this strategy gave us exactly the same result as the base

model, which indicates that probably none of the new words were in our dev data. So, after consideration we decided to leave it out too.

So now, we were ready to try predicts the tags based on previous information. First, we thought about predicting with a bigram of each two words. But then, we realized that we could use our information from the POS part. We have models that predict the POS tag with accuracy greater than 90% so we decided it's a great idea. Naturally, we first tried to use our best model, the one who use contextualized word embedding, but like our other experiments we found out its too heavy. Nonetheless, our basic Bigram Pos Model still achieved good result (~93%) and it's running fast se we used it. So, our new approach was first predicting the POS for each word in our data and then use it to predict the NER tag. We did it in two steps, predicting with the current word POS and also with the previous POS.

Predict with the current POS:

Overall Accuracy (including 'O'): 93.93%

	Precision	Recall	F1
All-types*	**70.80%**	**67.17%**	**68.94%**
MISC	73.25%	69.20%	71.17%
PER	68.61%	55.05%	61.08%
LOC	80.79%	81.06%	80.92%
ORG	58.82%	63.39%	61.02%

*excluding 'O'

Predict with the current POS and previous POS:

Overall Accuracy (including 'O'): 94.22%

	Precision	Recall	F1
All-types*	**72.55%**	**68.51%**	**70.48%**
MISC	74.86%	70.72%	72.73%
PER	69.34%	55.48%	61.64%
LOC	83.66%	82.20%	82.92%
ORG	60.71%	66.14%	63.31%

*excluding 'O'

After completing these steps, we reached the stage of handling out-of-vocabulary words (OOV). Opting to persist with our effective strategy of identifying the most similar words and predicting tags accordingly. We scrutinized the training data in comparison to the development data. It became evident that approximately one-third of the words in the development data were absent from our training dataset. Given this high rate of omissions, we held a strong belief that adopting this approach would substantially enhance performance. But, once again, it was too heavy in terms of running time. So, we had to think about out-of-the-box solutions. We started to identify patterns in the data and then built a rule-based pipeline that passed over our initial predictions again.
First, we identified some obvious patterns like numbers, dates, and more. After that, we recognized that words that have capital letters not in the beginning of the sentence are often NER's. A lot of these words are OOV so now it was a good idea to use the contextualized word model. We continue to reduce the size of the unknown words to a reasonable size by filtering to relevant POS, filtering some common words and the most important thing is to pass each word just once in the assumption that

missing words are rare, so they come often in the same context. Some more rules we found are that the words ‘of’ and ‘in’ that come after ‘ORG’, ‘MISC’, or ‘LOC’ are likely to get their previous word tag.

This approach improved our performance gradually and we got the following results:

Overall Accuracy (including ‘O’): 95.7%

	Precision	Recall	F1
All-types*	**73.83%**	**79.57%**	**76.59%**
MISC	74.12%	72.67%	73.38%
PER	75.82%	85.29%	80.28%
LOC	83.05%	85.36%	84.19%
ORG	59.68%	68.53%	63.80%

*excluding ‘O’

We analyze our result predictions after every improvement, trying to identify further opportunities for improvement. We have noticed that our performance on the ‘ORG’ tags is still low, so we have decided to focus on them. We found that many organizations use locations and other identifying information in their names (such as sports clubs or businesses starting with the city name like ‘New York Knicks’). Therefore, our idea is to iterate over each multiple-word entity and assign the most common tag to the entire entity, instead of assigning different tags to different words within the same entity. We have modified the model accordingly and got the following improved result!

Overall Accuracy (including ‘O’): 96.39%

	Precision	Recall	F1
All-types*	**81.20%**	**82.73%**	**81.96%**
MISC	84.21%	88.87%	86.48%
PER	70.37%	75.99%	73.07%
LOC	82.33%	74.30%	78.11%
ORG	86.07%	85.74%	85.90%

*excluding ‘O’

Although these approaches are helpful, they do have some limitations. For instance, we predict tags on a per-word and not per-entity basis, and we don't attempt to predict the "B" in the "bio" method. Therefore, when we modify entities as described above, we need to be aware that we could unintentionally combine two entities together. In addition, due to the utilization of the contextualized model, the running time increases significantly from a few seconds to several minutes.

To sum up, this is our best NER model.