Learning from Data – Assignment 4: Perceptrons & Word Embeddings

General remarks

This assignment is meant to get you acquainted with the basics of neural networks, by working with a perceptron classifier and word embeddings. There are three components: some theoretical questions, some exploration of word embeddings, and experimenting with the perceptron. You are provided with (the code for) a basic perceptron classifier, and will have to make adaptations to that.

What you have to hand in:

- Python code, which should show how you got your answers for questions 4.2.x, please call it
  `LFDassignment4_yourname.py`
- research report (based on the template) that describes what you've done and also includes answers/discussion to all questions in this document. Please, make sure you submit a `pdf` file.

Deadline: Wednesday, December 14th, 11.00pm

Data

For this assignment, we will be using a set of Named Entity Disambiguation data, extracted from the OntoNotes corpus. It consists of single-word named entities, labelled as belonging to one of six different categories: geo-political entity (GPE), location (LOC), person (PERSON), organisation (ORG), date (DATE) and cardinal number (CARDINAL). The data is in the file `named_entity_data.txt` on Nestor.

Exercise 4.1 – Word Embeddings

In class, you were shown a demonstration of two word2vec tools. One gives the most similar words, given a certain input word. The other does analogy, e.g. given the input *Paris France Berlin*, it will provide *Germany* as the top answer. In the following exercises, you'll have to explore this, and try to come up with interesting similarities and analogies.

The two tools, and the trained word embeddings, are available in `/net/shared/word2vec`. The word embeddings are in the file called `GoogleNews-vectors-negative300.bin`, the similarity tool is called `distance`, and the analogy tool is called `word-analogy`. To start the tool, just type: `./distance GoogleNews-vectors-negative300.bin`. This might take a short while to load, and requires at least 4.5 GB of free RAM.

You can run this on one of the LWP machines, via ssh, or download and install it yourself. On the LWP machines, just navigate to `/net/shared/word2vec`, and proceed from there.

To access this directory from a different machine, you can login through ssh on the students' server, Karora, and run it on there. Note that Karora has very limited resources, so running it with many students at the same time might cause problems.

To ssh from a Linux/OSX system, go to the terminal, type
`ssh -p 2222 s1234567@bastion.service.rug.nl`, (replace s1234567 with your own student number) and provide your RuG-password. This gives you access to the Bastion server, from which you can connect to Karora by typing `ssh karora.let.rug.nl`, and providing your password again.

To ssh from a Windows machine, the easiest solution is to download and use the SSH-client PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`, putty.exe). In the main screen of PuTTY, enter the server name `bastion.service.rug.nl`, and port number `2222`. Then, provide your student number and your password. This gives you access to the Bastion server, from which you can connect to Karora by typing `ssh karora.let.rug.nl`, and providing your password again.

If you prefer to do everything on your own machine, you can simply download the word2vec toolkit from `https://code.google.com/p/word2vec/`, and install it on your own machine. Make sure you also download the pre-trained word embeddings (`GoogleNews-vectors-negative300bin.gz`).

4.1.1 Similarities

If you try out some inputs in the distance tool, and look at the most similar words, you'll start to notice some oddities. If you type *Groningen*, for example, you'll see that it doesn't just return other Dutch cities, but also Dutch and foreign football teams, and a Dutch university. This is because *Groningen* is strongly associated with not just the city itself, but also its university and football team.

For this exercise, try out at least 5 different words in the similarity tool, and see if anything unexpected comes up, similarly to the example described above. For each case, report the input word, what you expected to be the most similar words, and, if the results are different than expected, what might be the cause of that. In your report, you don't need to give the full list of most similar words, just the interesting ones is sufficient. Interesting words to try are ones which have multiple meanings, or which have strong connotations.

4.1.2 Questions on word meaning ambiguity

1. For words which have multiple meanings, we distinguish two categories: *polysemous* words, and *homonymic* words. What is polysemy, what is homonymy, and wha is the difference between the two? Please give examples for both.
2. Word embeddings are especially suited for word-level classification tasks, such as part-of-speech tagging or named entity labelling. An exception to this is the established NLP task called *word sense disambiguation*. What is word sense disambiguation, and why are word embeddings (as given in this assignment), not a good feature for this task?
3. In the similarity tool, type *cookie*, and look at the most similar words. Then, start the same tool, but with another (smaller) set of vectors (`vectors.bin`), using `./distance vectors.bin`. Now try *cookie* again. What do you see? What could have caused this? Keep in mind how embeddings are trained and what they attempt to capture.

4.1.3 Novel analogies

In class, we saw that word embeddings can capture certain relations. For example, it can capture the male-female relation, by coming up with the vector for *queen*, given the input *man woman king*. The same goes for countries and their capitals, and countries and their demonyms.

In this exercise, try to come up with 3 new relations that word embeddings might be able to capture, and test these out using the analogy tool. Give the relations you tried to capture, 3 test cases (e.g. for the male-female analogy, try king-queen, actor-actress and monk-nun), and whether they were successful or not. If the relation was not captured successfully, try and come up with a possible explanation, if possible.

Please, try to come up with your own relations, and don't use any of the three examples mentioned in this assignment.

Exercise 4.2 – Using a perceptron for named entity disambiguation

You are given three files for this assignment:

- `named_entity_data.txt`: a corpus of single-word named entities, and their named entity labels. Each training example is on one line, which consists of a word, followed by its label, and separated by a tab.
- `embeddings.pickle`: a binarized version of a Python dictionary containing the 50-dimensional word vectors, taken from here: `http://nlp.stanford.edu/projects/glove/`. It is binarized, so it's quicker to read in.

  Note: this set of embeddings contains only lower-cased, single words (so *berlin*, but not *Berlin*), in contrast with the embeddings you used in 4.1, which contained both capitalized words and multi-word expressions.

- `LFDassignment4.py`: a Python script, similar to the ones from previous assignments, that does basic Perceptron classification with default parameters, and reads in the data and word embeddings.

  Note: you should run the script as follows:
  `python3 LFDassignment4.py <input_data_file> <embeddings_file>`

4.2.1 Baseline

Currently, the script you're given does a basic binary classification, classifying named entities as either 'LOCATION' (==GPE/LOC) or 'NON-LOCATION' (==PERSON/ORG/DATE/-CARDINAL). It does this with approx. 81% accuracy. This seems good, but by itself, does not mean much. It could be, for example, that the class distribution is very skewed, e.g. $80\%/20\%$. In that case, it is not very hard to give the right classification in 80% of the cases.

To put performance into perspective, we need a baseline measure to compare it to. Implement a simple baseline measure for the binary classification, and report accuracy, precision, recall and F1-score for the baseline and the classifier. Do the same for the six-way classification. Do you still think the perceptron performs well?

4.2.2 Perceptron Parameters

The perceptron currently uses default parameters. Of course, there is no reason why these parameter values should yield optimal performance. In this exercise, experiment with different parameter values, and see what their effect on (both binary and six-way) classification performance is. The only parameters you are required to change are `n_iter`, `learning_rate` and `penalty`, but of course you are encouraged to try other ones, as well. Keep the value for `random_state` the same, to make sure your results are comparable.

4.2.3 Word Embeddings and Generalization

As you have seen, word embeddings are very good at capturing similarities between words. The advantage of this for NLP is that it allows a model to generalize from the words in the training data towards similar, but unseen words.

In this exercise, try to see if the perceptron manages to learn some generalizations in 6-way classification. Do this by having the classifier classify a word that is not in the training data, but that does have words similar to it in the training data. Then, check if this word gets classified correctly. Do this for 3 different words. Report the following: the word you tested (e.g. *Hannover*), some similar words in the training data (e.g. *Berlin*, *Hamburg*, *Bremen*), their assigned class (e.g. GPE), and whether or not it got classified correctly.

4.2.4 Error analysis

If you want to know why the classifier performs as it does, it is useful to take a look at the errors it makes, and see what causes them.

To look at the errors, add the code to produce the confusion matrix (for six-way classification, naturally). Report which classes get confused the most, and which get confused the least. Take into account that some classes occur much more often in the test data than others; 40 misclassifications out of 2000 is not a lot, 40 out of a 100 instances is. Then, try to explain why it confuses some classes, and not others. Think about what type of named entity each label denotes, how words (==embeddings) are located in the vector space, and what kind of classifier a perceptron is.