INF5820, Fall 2018
Assignment 2: Distributional Word Embeddings

UiO Language Technology Group

Deadline 5 October, at 22:00, to be delivered in Devilry

Goals

1. Learn how to use *Gensim* library to deal with word embeddings in Python.
2. Learn how to evaluate pre-trained word-embedding models.
3. Get experience in training a neural document classifier using word embeddings as building bricks for feature sets.

Introduction

This obligatory assignment aims at using dense word embeddings in typical natural language processing tasks. In it, you will implement a document classifier based on a feed-forward neural network, using **continuous bags-of-words** (combination of dense vectors for words) as features. Primary and recommended machine learning framework in this course is *Keras* with *TensorFlow* as a backend. These are provided out-of-the-box in the LTG environment on the Abel cluster[1], and we strongly encourage you to train your models using Abel. If you would like to use any other deep learning software library, it is absolutely allowed, provided that your code is available.

Please make sure you read through the entire assignment before you start. Solutions must be submitted through Devilry[2] **by 22:00 on October 5**. Please upload a single 4-8 pages PDF file with details on your experiments and your answers to the questions posed in the assignment.

Your (heavily commented) code and data files should be available in a separate private repository in the UiO in-house Git platform[3] (we remind that the official programming language of the INF5820 course is *Python 3*). The code must be self-sufficient, meaning that it should be possible to run it directly on the data. If you use some additional data sources, these datasets should be put in the repository as well. If you use some non-standard *Python* modules, this should be documented in your PDF report. Make the repository private, but allow full access to INF5820 teaching staff[4]. The PDF in Devilry should contain a link to your repository.

[1] `https://www.uio.no/studier/emner/matnat/ifi/INF5820/h18/setup.html`
[2] `https://devilry.ifi.uio.no/`
[3] `https://github.uio.no/`
[4] `https://github.uio.no/orgs/inf5820/people`

If you have any questions, please raise an issue in the INF5820 Git repository[5] or email *inf5820-help@ifi.uio.no*. Make sure to take advantage of the group sessions on September 27 and October 4.

Recommended reading

1. **Neural network methods for natural language processing.** Goldberg, Y., 2017.[6]
2. **Speech and Language Processing**. Daniel Jurafsky and James Martin. 3rd edition, 2018. Chapter 6, 'Vector Semantics' [7] (sections 6.2-6.4, 6.8-6.12)
3. **Word2vec parameter learning explained**. Xin Rong, 2014.[8]
4. **SimLex-999: Evaluating Semantic Models with (Genuine) Similarity Estimation**. Felix Hill et al, 2015.[9]
5. Linear Algebra cheat sheet by the LTG[10]
6. `https://radimrehurek.com/gensim/models/word2vec.html`
7. `https://keras.io/`
8. `https://www.tensorflow.org/`
9. `http://research.signalmedia.co/newsir16/signal-dataset.html`

The assignment is divided into 4 parts:

1. **Basic operations with word embeddings**;
2. **Intrinsic evaluation of pre-trained word embeddings**;
3. **Training a word embedding model on in-domain data**;
4. **Document classification with word embeddings**;

1 Basic operations with word embeddings

Prediction-based distributional semantic algorithms (also known as **word embedding** algorithms) work under the hood of the majority of intelligent systems dealing with natural language. They became especially popular after the introduction of **Continuous Bag-of-Words** and **Continuous Skip-gram** algorithms, first implemented in the famous *word2vec* tool. Their ultimate aim is to learn meaningful vectors (embeddings) for words, such that semantically similar words have similar vectors. It can be profitable to try to implement a

[5] `https://github.uio.no/inf5820/course2018/issues`
[6] `https://www.morganclaypool.com/doi/10.2200/S00762ED1V01Y201703HLT037`
[7] `https://web.stanford.edu/~jurafsky/slp3/6.pdf`
[8] `https://arxiv.org/pdf/1411.2738.pdf`
[9] `http://www.aclweb.org/anthology/J15-4004`
[10] `https://www.uio.no/studier/emner/matnat/ifi/INF5820/h18/timeplan/la.pdf`

neural embedding algorithm from scratch (we encourage you to do this); however, in real life one usually employs existing implementations and trains own models with them, or even re-uses some of the available pre-trained models. In this part of the obligatory assignment 2 you will get familiar with some of this software. We suggest *Gensim* library, but you are free to use other existing tools. We provide some starter code to work with *Gensim*[11].

1.1 WebVectors: word embeddings online

Make yourself familiar with the *WebVectors* web service maintained by the LTG Oslo group[12]. You don't need to install or download anything. *WebVectors* features pre-trained word embedding models for English and Norwegian (and some other languages, if you follow the links to the NLPL word embeddings repository). You can produce nearest semantic associates for any given word (with filtering by parts of speech), solve word analogies, calculate cosine similarity between pairs of words, plot words in the reduced 2-dimensional space, etc.

1.2 Working with pre-trained models locally

Web services are good for quickly demonstrating a technology, but for the majority of real-world tasks you would like to have models at your hands, especially if you have lots of data to process. We will now work locally with the models under the hood of *WebVectors*. Download the pre-trained English models for Wikipedia and Gigaword, listed at `http://vectors.nlpl.eu/explore/embeddings/models/` (model identifiers **3** and **29**, correspondingly). They are published in the standard NLPL repository format: as a `zip` archive containing `README` file, `meta.json` file with the model metadata, and the word embedding model itself as `model.txt`.

These models can be easily loaded into any software able to work with word embeddings. Try the example *Gensim* code[13]. You can run the script with the model filename as an argument:

```
python play_with_model.py 3.zip
```

It will load the chosen model (without decompressing the archive contents to disk) and invite you to enter a query word. If the word is present in the loaded model, its 10 nearest associates together with their cosine similarity to the query word will be printed. If you enter several words separated by spaces, the model will try to find out which of them is the most semantically distant from the others ('doesn't belong here'): for example, '*fire*' doesn't belong to the list '*orange, apple, pineapple*'. Note that these models contain lemmatized words augmented with their part of speech tags in the **Universal Dependencies** tagset[14] ('*parliament_NOUN*', etc). Find out which *Gensim* methods the code uses (you can consult the manual from the **Recommended reading** section). You should be most interested in the `most_similar()` method.

Your task in this part of the assignment is to analyze the content words from the first sentence of the abstract to the Chen and Manning '*A Fast and Accurate*

[11] `https://github.uio.no/inf5820/course2018/tree/master/lab_27_09`

[12] `http://vectors.nlpl.eu/explore/embeddings/`

[13] `https://github.uio.no/inf5820/course2018/blob/master/lab_27_09/play_with_model.py`

[14] `http://universaldependencies.org/u/pos/`

Dependency Parser using Neural Networks' paper[15]. Particularly, you are to use the English Wikipedia and Gigaword models to find the not-so-nearest semantic associates for these words: not the first 10 associates (those can be easily found using the *WebVectors*), but the next 5.

1. Modify the provided script (or write your own) so that it is able to take a text file with query words (one word per line) as an input;
2. create such an input file with all the (lemmatized and PoS-tagged) content words from the first sentence of the abstract;
    - you are free to either use any of the available taggers or simply lemmatize and tag the words manually (the sentence is not that long).
3. modify the script so that for each input word it outputs its **11^{th}, 12^{th}, 13^{th}, 14^{th} and 15^{th} nearest semantic associates** (with the corresponding cosine similarities to the query word);
4. save the produced associates and similarities into a file.
5. Are there any notable differences in the results from the Wikipedia and Gigaword models?

2 Intrinsic evaluation of pre-trained word embeddings

As a rule, you are interested in discovering how good is the word embedding model you are using. If for some reason you can't evaluate it on the downstream task, you can still use the so called **intrinsic** evaluation methods, which hopefully correlate with the prospective practical performance of your system.

One of the established methods to intrinsically evaluate distributional models is to measure how good they are in reproducing human experts' judgments about **semantic similarity between pairs of words**. The idea is that we ask a significant amount of human annotators to rank word pairs according to their similarity and calculate cosine similarities for these pairs in the model under evaluation. Then we simply measure the Spearman rank-order correlation coefficient[16] between the two lists of similarities. Correlation close to **1** means that the model is extremely good in mimicking human judgments, and thus is supposedly superior in most downstream tasks. There are many published semantic similarity datasets, and lots of academic discussion goes on around them.

Another popular intrinsic evaluation method is to use the so called **analogy datasets**: manually created quadruplets or proportions of semantically related words, in which a model has to guess the last element. For example, for the sequence '**Paris, France, Oslo, ???**' the model should output **'Norway'**, thus making an analogical inference related to the notion of being a capital of a country. The performance is measured as simple accuracy (ratio of correct answers). Analogy datasets evaluate the embeddings' ability

[15] `http://www.aclweb.org/anthology/D14-1082`

[16] `https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient`

to capture complex semantic relations between 'constellations' of words. *Gensim* offers methods for evaluating word embedding models both using semantic similarity datasets (`evaluate_word_pairs()`) and using analogy datasets (`evaluate_word_analogies()`).

Your task is to evaluate several word embedding models and find out the correlation between their performance in semantic similarities and in analogical inference.

1. Download the evaluation datasets:
   - `https://github.uio.no/inf5820/course2018/blob/master/obligatories/2/simlex.tsv`. This is the popular **SimLex999** semantic similarities dataset (see [Hill et al, 2015] from the **Recommended Reading** section);
   - `https://github.uio.no/inf5820/course2018/blob/master/obligatories/2/analogies_semantic.txt`. This the so called **Google Analogies** dataset first proposed in [Mikolov et al, 2013]. We removed all the 'syntactic' sections from it, leaving only the 'semantic' ones.
2. Visually inspect the files, make sure you understand their format.
3. Visit the NLPL word embedding repository[17] and download the models with the following identifiers:
   - **40**. It was trained on the English *CoNLL17* corpus, using *Continuous Skipgram* algorithm with the vector size 100, and the window size 10.
   - **75**. It was trained on the English *Oil and Gas* corpus, using *Continuous Bag-of-Words* algorithm with the vector size 400, and the window size 5.
   - **82**. It was trained on the English *Common Crawl* Corpus, using *GloVe* algorithm with the vector size 300, and the window size 10.
4. Evaluate these models, using the datasets mentioned above as the gold standard.
5. Evaluate the models separately on each section of the **Google Analogies** dataset ('capital-common-countries', 'gram1-adjective-to-adverb', etc), and on each PoS section of the **SimLex999** dataset (noun, adjectives, verbs).
6. Save the produced evaluation scores into a file, describe them in your report.
7. Is there any correlation between the models' scores on semantic similarity and analogy evaluations? Are there any sections of the datasets, where this correlation is stronger or weaker?
8. Provide possible explanations for the observed phenomena.

[17] `http://vectors.nlpl.eu/repository/`

3 Training a word embedding model on in-domain data

NLP practitioners often use pre-trained word embedding models created by someone else, in the hope that these models will be good enough for their task at hand. However, in some cases (particularly if your data is highly specific), it might make sense to **train your own word embedding model on your in-domain data**. Since in the next section of this assignment you are going to build a document classifier for the *SignalMedia* texts, it is worth trying a word embedding model trained on all of these documents (along with other pre-trained word embeddings).

The Signal Media One-Million News Articles is the same dataset you already saw in the obligatory assignment 1. It contains texts from news sites and blogs published in September 2015. Recall that the texts are already lemmatized and PoS-tagged. Thus, a sentence '*Satellite will play an important role in the fight against illegal fishing*' is converted to '*Satellite_ NNP play_ VB important_ JJ role_ NN fight_ NN illegal_ JJ fishing_ NN*'. Note that the *Penn Treebank* (PTB) tag set was used for part of speech tagging[18] and that unlike in the previous task, the word case is now preserved.

In the obligatory assignment 1, you worked with a part of *SignalMedia*: texts published on 10 particular web sites. Of course, the whole corpus is much larger. In the file `/projects/nlpl/teaching/uio/inf5820/2018/signal_texts.txt.gz` (accessible from Abel nodes), you will find all texts from the the news part of the *SignalMedia* (excluding blog posts). It contains about 163 million word tokens after stop words removal. This is not much compared to gigantic corpora like *Google News*, *CommonCrawl* or *Gigaword*, but still enough to train a good word embedding model. What is important is that this model will (hopefully) capture important semantic properties of words in this particular type of texts.

The corpus is provided as a gzipped plain text file with one document per line, words separated with spaces. Note that there are no source labels in it, so it can't be used to train classifiers: it is only a source of co-occurrence data to train word embedding models. Your task is to train such a model.

1. **Train a word embedding model** on the provided *SignalMedia* texts.
   - You can use the provided starter code[19].
   - Choose any hyperparameters you like, but the resulting model should have at least 100 000 words in its vocabulary.
2. Train another model on the same data, but with one hyperparameter different (for example, window size or vector size).
3. The trained models are saved in the native *Gensim* format, but can be converted to the standard word2vec formats (see the example conversion script[20]).

[18] `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

[19] `https://github.uio.no/inf5820/course2018/blob/master/lab_27_09/train_word2vec.py`

[20] `https://github.uio.no/inf5820/course2018/blob/master/lab_27_09/convert.py`

4. Together with the 3 *WebVectors* models you downloaded in the previous section, you now have 5 word embedding models.

5. Use them to compare the lists of **top 20 most frequent words** in *SignalMedia*, *CoNLL17*, *Oil and Gas*, and *Common Crawl* corpora.

   - In *Gensim*, one can access the model vocabulary sorted by frequency as `MODEL.index2word` list.
   - If a model is saved in the native *Gensim* format, one can also access the words frequencies in the training corpus (as integers) with the `MODEL.wv.vocab['YOUR_WORD'].count` method.
   - Note also that if a model is saved as a text file in the word2vec format, its lines are as a rule sorted by frequency as well. Thus, top 20 lines of the file correspond to 20 most frequent words in the training corpus.

6. Are the lists different? What does it tell you about the nature of these corpora?

7. **Intrinsically evaluate** both *SignalMedia* models on the test sets you worked with in the previous section (*SimLex999* and *Google Analogies*).

   - Note that the test sets are not PoS tagged, while your training corpus is. Since you probably don't want to lose the information contained in PoS tags, you'll have to modify the test sets. Analyze them and find a way to automatically add proper PoS tags so that they correspond to the tags in the *SignalMedia* corpus. Save the modified test sets in your repository.

8. Do you see any difference in the performance of 2 models? How in your opinion it is related to the changes you've made to hyperparameters?

9. How the intrinsic performance of the models you trained compares to the performance of the pre-trained embeddings downloaded from *WebVectors*? Give possible reasons for your findings.

4 Document classification with word embeddings

Distributional semantic models can represent entities larger than words: phrases, sentences and whole documents. This makes it possible to train semantically-aware document classifiers. In this task, you will have to train such a classifier using deep neural networks and the same English word embedding models that you got familiar with before.

The classification task itself is very similar to the one in the obligatory assignment 1. The training data is in the same format and can be found at `https://github.uio.no/inf5820/course2018/blob/master/obligatories/2/train_signal_10_oblig2.tsv.gz`. For the sake of the task being a little more interesting, we replaced two of the news sources with other ones, so now the list of classes looks like this:

1. MyInforms
2. Individual.com

3. 4 Traders
4. NewsR.in
5. Reuters
6. Mail Online UK
7. App.ViralNewsChart.com
8. Latest Nigerian News.com
9. **EIN News**
10. **Yahoo! News Australia**

There are 49 090 documents in the training data. This time, we also have a held-out test set of 5 455 documents, which **will not be published until we receive all the submissions**. Then, your solutions will be evaluated on this test set, so that we can rank the systems by their objective performance. The task is again to predict the document class (source) based on the words occurring in this document.

In the assignment 1, you did this by representing all the documents as **bags-of-words**: that is, extracting the collection vocabulary (the union of all word types in the texts) and counting frequencies of each word type in each document. Documents were then represented as sparse vectors of the size equal to the size of the vocabulary. But this time, we would like to:

1. leverage **semantic information** (treat semantically similar words similarly);
2. work with **dense low-dimensional document vectors**, not with sparse high-dimensional vectors produced by the bag-of-words approach.

Your task is to come up with semantically-aware representations of the documents in the training dataset and to create classifiers able to predict the document class using these representations. The classifiers (again) are supposed to be feed-forward neural networks, but this time they should take as an input not one-hot word representations, but continuous word embeddings.

You can safely use the code for the classifiers from the assignment 1 (or from the example solution[21]). The necessary change will be to somehow produce meaningful **dense** representations of the documents, instead of BoW. You can implement any approach to that that comes to your mind. The most straightforward way is to use the so-called **semantic fingerprints** algorithm (averaging or summing up the embeddings of words in the document). You can either:

1. Transform word sequences into 'semantic fingerprints' **beforehand**, and then feed the resulting averaged vectors as features into your neural model.
    - For that, implement the `fingerprint()` function in `https://github.uio.no/inf5820/course2018/blob/master/obligatories/2/helpers.py`. It should lookup vectors for document words in the given word embedding model, and then return the sum or the average of these

[21] `https://github.uio.no/inf5820/course2018/tree/master/obligatories/1/solution`

vectors, skipping the words which are not present in the model. It should support both 'binary' and 'count' modes (using either sets or lists of document words).

2. Use the pre-trained word embedding model as an `Embedding()` layer in *Keras*, and then **perform the summation or averaging of the resulting vector arrays for each document in the computation graph itself**, using the *Keras* bult-in functions (`Add()`, `Average()`, etc). Note that *Gensim* supports direct conversion of loaded word embedding models into *Keras* layers via the `get_keras_embedding()` method.

Since this time there is a separate test set (not yet published), you don't have to separate the data into the training and test parts. You still have to extract some part of the data as a development/validation set, either beforehand in a separate file, or using the `validation_split` parameter in *Keras*.

4.1 Training a classifier

Train a fully-connected **feed-forward neural network** classifier on the semantic fingerprints produced from the documents in the training dataset, evaluating it on the development corpus. Experiment with different activation functions, layer dimensionalities and regularization terms. Because you use pre-trained word embedding models with the fixed vocabularies, you don't need to extract vocabulary from the text yourselves (no need for the `Tokenizer()` and the like).

Since your document representations now depend on what word embedding model you are using, it is of utmost importance to find out which of them is better for this task. Try all the *WebVectors* models you played with in the previous sections of this assignment (identifiers **3**, **29**, **40**, **75**, **82**), and the models you trained on the whole *SignalMedia* corpus. Note that in different models, different PoS tagging schemas are used. In the models **3** and **29**, words are augmented with the **Universal Dependencies** (UD) PoS tags. Thus, to use them, you'll have to implement the `tag_convert()` function in `https://github.uio.no/inf5820/course2018/blob/master/obligatories/2/helpers.py`, which should take as an input a word with the PTB PoS tag and return the same word with the corresponding UD PoS tag. The models **40**, **75** and **82** do not contain PoS tags at all, so you'll have to simply strip them from the texts, if you use these models. The models trained on *SignalMedia* are of course fully consistent with the training corpus format, so you don't have to do anything in this case.

Describe the parameters of your network and the evaluation results in the PDF report. List the performance of the classifiers using different word embedding models. Why some of them are better and some of them are worse (with the same neural architectures)? Can you think of some set of hyperparameters to train word vectors on the *SignalMedia* corpus, which can be particularly useful for this classification task? Train a word embedding model using these hyperparameters and test your hypothesis.

As an additional experiment, implement a version of the classifier which does not rely on any pre-trained word embedding. It should extract the vocabulary of the desired size from the training set (as in the obligatory assignment 1), and initialize an `Embedding()` layer with random vectors for the extracted words. These vectors are then considered to be the parameters of the neural network and are **trained along with other weight matrices**, thus optimizing word

embeddings for this particular classification task. Report on the changes in classification performance and in the training time with this approach.

We remind that you should report the **macro F1 score**. Find your best-performing classifier and save it as a *Keras* model. If you use a custom word embedding model (not a model from *WebVectors*), save it as well. Recall that probabilistic classifiers sometimes can produce different results with the same hyperparameters because of different random initializations. Thus, train your best classification architecture 3 times and evaluate it 3 times, reporting the average and the standard deviation.

4.2 Comparing against BoW classifier

Once you have chosen your best-performing classifier, compare it with the **bag-of-words** approach you used in the previous obligatory assignment (set the vocabulary size to 3 000). Use the same neural architecture, change only the input features. Vary the number of hidden layers and analyze how the F1 performance of the classifier changes when using BoW and when using word embeddings. What approach is better with regards to the classification performance and with regards to the training time? What are theoretical memory requirements for BoW and for dense representations of documents?

Give your opinion on whether it is worth using word embeddings in this particular document classification task.

5 Conclusion

Your best classifier should be published in your UiO Github repository as a saved *Keras* model. If this best classifier used a word embedding model you trained on the *SignalMedia* texts, this model must be published as well.

Also, you have to provide an `eval_on_test.py`[22] script. It should take as an input a saved model, word embedding file and a test dataset, and return accuracy, precision, recall, and F1 scores for each class in the test set, as well as their macro average values. We will use it to evaluate your classifier on the held-out test set. Hope your system will be the best!

Good luck and happy coding!

[22]Similar to the one at `https://github.uio.no/inf5820/course2018/blob/master/obligatories/1/solution/eval_on_test.py`