Big Data Science

—

Neural word embeddings for NLP

Learning outcomes

- Training neural word embeddings on a subset of the *Project Gutenberg* corpus.
- Using word embeddings to predict semantically similar words.
- Using word embeddings and vector arithmetic to solve word analogies.
- Visualizing the embedding space using t-distributed stochastic neighbor embedding (t-SNE).

A word of advice: Model training can take up to a few hours, so start this practice early. If you have access to HPC, it might be a good idea to set this up on the cluster instead of your local machine.

Getting started

For this practice, we will use Deeplearning4j (DL4J), an open-source deep learning programming library for Java and Scala that also runs on Hadoop and Spark.

Prerequisites

Please refer to the DL4J Quick Start Guide to get set up.

All prerequisites are listed in the Prerequisites section of the quick start guide.

Most importantly, you will need:

- Java 1.7 or later
- Apache Maven
- a suitable IDE (IntelliJ or Eclipse)
- Git

Again, please refer to the DL4J Quick Start Guide for installation instructions.

Next, take a look at the DL4J Examples in a Few Easy Steps. To install, run:

```
$ git clone https://github.com/deeplearning4j/dl4j-examples
$ brew install maven
$ mvn clean package
```

Please follow the Quick Start Guide to complete the installation and run a few examples.

Dataset

For this practice, we will use the Gutenberg Dataset, a collection of 3,036 English books written by 142 authors. All text files have been concatenated to create one single data file that we will use as input to the *word2vec* model (1.21GB total).

Download the data file here, then unzip the archive to extract the .txt file:

https://drive.google.com/file/d/1wBpgETqT6TCzyzB70B9n12Etq8tabk6g/view?usp=sharing

Afterwards, it's easiest to just move the file you downloaded to the DL4J resources directory (*.../dl4j-examples/dl4j-examples/src/main/resources/*) so you can specify the path to this file as your *ClassPathResource* when loading the data later on.

Tasks

Understanding word embeddings

One of the most popular tools to generate neural word embeddings is called *word2vec*.

To understand what word embeddings are and how they are generated, please read the original research paper on the "Efficient Estimation of Word Representations in Vector Space" published by Mikolov et al. (2013). If you're interested, here is a recent survey on "Improving Distributional Similarity with Lessons Learned from Word Embeddings" (Levy et al., 2015) as well.

Next, read the Introduction to Word2Vec on the DL4J website. If you want, go ahead and read the rest of the tutorial as well, it will be useful for this practice.

Training word embeddings

Example code for how to train your own word embeddings using *word2vec* is provided in the Just Give Me the Code section of the DL4J tutorial.

Just as described, you will need to load and tokenize the data before you can train the model.

Be sure to train your word embeddings on the dataset we provided.

Use the following model parameters so we can easily evaluate your results[1]:

```
Word2Vec vec = new Word2Vec.Builder()
        .minWordFrequency(15)
        .iterations(1)
        .layerSize(200)
        .seed(0)
        .windowSize(5)
        .iterate(iter)
        .tokenizerFactory(t)
        .build();
```

It takes a while to train the *word2vec* model, so make sure to save the model file and reload it instead of retraining the vectors every time.

If you're running into any issues related to memory usage, refer to the Troubleshooting & Tuning Word2Vec section of the DL4J tutorial.

[1] After you submitted your results, you're encouraged to experiment with these parameter settings!

Predicting semantically similar words

Now that you have a trained model, we can evaluate the quality of the word vectors. All the code you will need for the following two exercises below can be found in the DL4J tutorial.

We will start by predicting similar words, using the built-in *wordsNearest()* functionality.

For each of the words below, please submit the top 10 most similar words:

cat
animal
science
scientific
vector
vendor
car
hot
major
man
doctor
flower
capital
washington

Briefly explain the results that seem surprising or problematic to you, if any.

Keep in mind which data source we used as our training data.

Solving word analogies

Next, let's take a look at word analogies.

For each of the following analogies, please submit the top 5 values of x:

king::queen	=	**x**::woman
paris::france	=	**x**::italy
car::cars	=	**x**::birds
man::woman	=	**x**::nurse
fall::stand	=	**x**::live
important::unimportant	=	**x**::unwilling

Briefly explain the results that seem surprising or problematic to you, if any.

Visualizing the embedding space

Visualizing your word embeddings is often helpful to gain a better understanding of your vector space. We will use DL4J's implementation of t-distributed stochastic neighbor embedding (t-SNE) to reduce the dimensionality of the word vectors and generate a 2D plot.

Since our vocabulary size is pretty large (over 100k unique words), we are only going to visualize a small subset of the vocabulary. Specifically, we are going to use the vocabulary of the SimLex-999 corpus, a gold standard resource for the evaluation of models that learn the meaning of words and concepts.

Download the deduplicated and sorted list of the corpus here:

https://drive.google.com/file/d/1yuOs2wmNZqFI2ocz0urRr8xAmc5GVdEA/view?usp=sharing

Afterwards, place the file into the root directory of your *dl4j-examples/* repository.

Unfortunately, the visualization examples on the DL4J website are outdated and will not work. **Instead, please follow the steps below to get started on generating your plot.**

First, load the *word2vec* model file you saved during training:

```
// Load model
Word2Vec wv = WordVectorSerializer.readWord2VecModel("path_to_model");
```

Our external word list might contain words the model never saw during training. Therefore, we need to exclude unknown words from plotting:

```
// Get model weights
WeightLookupTable weightLookupTable = wv.lookupTable();

// Get model vocabulary
VocabCache vocabCache = weightLookupTable.getVocabCache();

// Get set of known vocabulary words
Set<String> known = new HashSet();
for (Object s: vocabCache.words()) {
    known.add(s.toString());
}

List<String> words = new ArrayList<>();

// Read external word list
Scanner s = new Scanner(new File("simlex999_uniq_sort.txt"));
```

```
while (s.hasNext()){
    String tok = s.next();
    if (known.contains(tok)) {
        words.add(tok);
    }
}
s.close();
```

Next, we need to create a new matrix that only contains the vectors for the words in our list and convert it to an *INDArray* object that can be passed into the t-SNE builder:

```
double[][] doubleMatrix = new double[words.size()][];

for (int i=0; i<words.size(); i++) {
    doubleMatrix[i] = wv.getWordVector(words.get(i));
}

INDArray matrix = Nd4j.create(doubleMatrix);
```

Now we can train t-SNE as follows:

```
BarnesHutTsne tsne = new BarnesHutTsne.Builder()
    .theta(0.0)
    .build();

tsne.fit(matrix);
```

Note that we are using default parameters except for *theta*, the Barnes-Hut tradeoff parameter. *theta* typically ranges from 0-1, where higher values give a faster but less accurate optimization. A few empirical runs over our data gave the best results when not using the Barnes-Hut algorithm at all[2] and instead falling back to traditional t-SNE. Thus, *theta* is currently set to 0.

Feel free to experiment with the t-SNE parameters, but make sure the results are better or at least comparable[3] before submitting.

Currently, it is not possible to upload our results directly to the DL4J UI server. Instead, we will save the output in a .csv file, spin up the DL4J UI server, and upload the file manually:

```
tsne.saveAsFile(words, "tsne.csv");

UIServer uiServer = UIServer.getInstance();
StatsStorage statsStorage = new InMemoryStatsStorage();
uiServer.attach(statsStorage);
```

[2] I suspect there is a bug in the DL4J implementation somewhere that is related to numerical stability.
[3] o.d.plot.Tsne - Iteration [999] error is: [1.6632050275802612]

Once up and running, open a browser window and head over to http://localhost:9000/tsne.

From there, select “Choose file”, open your .csv output file, and click “Upload file”.

Your plot should be visible in the UI now. Each data point is a unique word from our external word list, annotated with its label. You can zoom in and out to inspect your results.

Please submit a snapshot of the word cluster surrounding each of the following words:

bird
brother
cheek
happy
music
salad
[choose another word yourself]

Hint: You should be able to simply search for these terms in your browser window and they will be highlighted in your plot.

Problems with word embeddings

By now you should be convinced that word embeddings are a really powerful tool. But, as you might have noticed during your experiments, there are cases in which our word embeddings don’t quite perform as expected.

For example, while word embeddings do capture word similarity, they also tend to go beyond just synonymy and include hypernyms and even antonyms of each word in the vocabulary as well. If you think back to the text mining homework, where the task was to simply extract keywords and common phrases for each document folder, integrating word embeddings in this case could actually be detrimental to the overall performance of your model.

More importantly, word embeddings also tend to amplify biases that exist in the training data, e.g. with regard to gender or racial stereotypes. Recently, there has been a lot of work in the NLP community that focused specifically on how to reduce such bias in word embeddings.

One particularly successful approach was published in “Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings” (Bolukbasi et al., 2016).

Please read the paper and briefly summarize the algorithms they used for debiasing word embeddings (in plain English).

Submission

Deliverables

Your submission should consist of the following:

- A plain text or .pdf file with all of your written answers and model outputs if applicable.
- The snapshots of your visualized word embeddings.

Grading

Deliverables	**Points**
Output for similar words	14
Output for word analogies	12
Snapshots of visualized embeddings	14
Summary of debiasing algorithms	10
(total)	**50**