DD2418 Language Engineering Johan Boye, Dmytro Kalpakchi
Assignment 3 2021-04-16

All programs should be written in Python 3, unless specified otherwise in the problem instructions. Don't use any external libraries (that are not part of the Python 3 distribution) unless otherwise specified.

Mandatory part

1. (Random Indexing) This assignment is dedicated to the exploration of the distributional word embeddings, a.k.a. word vectors. In the first problem we will explore *Random Indexing*. Your main task is to **extend the random_indexing.py script to make it create distributional word embeddings from the 7 books about Harry Potter**.

   Effectively your assignment consists of the following three tasks:

   1. **Clean the raw text**. The books about Harry Potter are provided as plain unformated text files, which should not be used directly for training any word vector model. For example, the text can be as unformated as the following snippets.

      ```
      1
      HARRY POTTER
      AND THE CHAMBER OF SECRETS
      by
      J. K. Rowling
      (this is BOOK 2 in the Harry Potter series)
      Original Scanned/OCR: Friday, April 07, 2000
      v1.0
      (edit where needed, change version number by 0.1)
      C H A P T E O N E
      THE WORST BIRTHDAY
      ```

      Having a word embeddings generation problem in mind, one can identify multiple problems with this text:

      - there is some unrelated text like `Original Scanned/OCR`;
      - there are some formatting artefacts, like the word `C H A P T E O N E` or sentences being broken by the newline character;
      - the punctuation is glued together with words, like `needed,` (why is it a problem?).

      For this assignment we will disregard the first two problems (what does it mean for the created word vectors?). Your task is to implement a `clean_line` function returning a line without punctuation and numeric symbols.

      To pass: You need to run `check_cleaned_text.sh`, which outputs differences between your cleaned text and the correct one. You should get no differences, i.e. empty console, after running the script.

   2. **Create word vectors**. Write the code creating word vectors using the Random Indexing technique. This would involve two steps: building a vocabulary of words which you are to embed and using Random Indexing to create word embeddings.

For the purpose of this assignment the vocabulary should contain every word present in any of the 7 provided Harry Potter books. **When creating word vectors, assume that the left context of the first word and the right context of the last word is empty.**

To pass: You should be able to call `get_word_vector` function and get a word vector for the word if it exists in the vocabulary and `None` otherwise.

3. **Find the closest words**. Write the code finding the closest words to the given ones in the induced vector space using a simple k-nearest neighbours algorithm (we suggest using `scikit-learn's` implementation of kNN algorithm).

   To pass: You should be able to call `find_nearest` function with a list of words of interest and get a list of the 5 nearest words with similarities. You should also be able to answer the following questions. What similarity metrics can you use in your algorithm? Which one would you prefer to use? Why?

   Try to find nearest neighbours for the following words:

Harry, Gryffindor, chair, wand, good, enter, on, school

   To give an example, our implementation returns the following 5 nearest neighbors for the word *Harry*: *Hagrid, Snape, Dumbledore, Hermione*.

   Experiment by changing various hyper-parameters of the Random Indexing algorithm, for instance:

   - change the dimensionality of the vectors to 10 with 8 non-zero elements (try different dimensionalities and number of non-zero elements);
   - change window sizes to the values of 0, 3, 10 making left and right windows both symmetric and asymmetric;

   How did the result change with the various modifications? What properties of the vectors do you expect to observe when setting left (right) window size to 0? Did you manage to observe these properties?

2. (word2vec) In this problem we'll explore a method called *word2vec* originally published in (Mikolov et al., 2013). Your main task is to **extend the w2v.py script to make it train distributional word embeddings (using word2vec method)** from the Harry Potter corpus. We require only that you train on the first Harry Potter book, but the results will be much better if you train on all 7 books (in which case the training will take longer).

   1. **Clean the raw text**. The text should be cleaned in the same way as for Random Indexing, i.e. you can just simply copy your `clean_line` and `text_gen` methods to the `Word2Vec` class.
   2. **Prepare data for Skip-gram training**. We will train word2vec using a skipgram formulation, i.e. trying to predict context words given the focus word. The preparations include implementing the `skipgram_data` method consisting of three main parts:

      - creating a vocabulary of words, for which the word vectors are to be created. In this problem we simply take all words without applying any kind of filtering. We recommend creating two mappings:

    - `w2i` converting words as strings to unique indices;
    - `i2w` converting indices back to words;
  - calculating two distributions over the words in the corpus:
    - a unigram distribution P_u;
    - a corrected unigram distribution P_s, defined by Mikolov et al. (2013) for negative sampling (see lecture slides or original article for more details);
  - creating lists of focus words and respective context words (for this you'll need to implement a `get_context` function) and returning both of these lists.

  When processing each line of the file, assume that the left context of the first word and the right context of the last word is empty.

3. **Training word2vec model**. The last step is to simply train word2vec model and here you have to implement 2 methods:
  - `negative_sampling` for performing a negative sampling according to (Mikolov et al., 2013). Experiment with sampling using the proposed distribution P_s and the regular unigram distribution P_u. Which one works better for you?
  - `train` for performing a gradient descent training. Experiment with a number of hyper-parameters:
    - weight initializations - try sampling from a uniform or normal distributions with different parameters;
    - learning rate scheduling - try implementing the one used in the original word2vec implementation, i.e.:

$$\alpha = \begin{cases} \alpha_{\text{start}} \cdot 0.0001, & \text{if } \alpha < \alpha_{\text{start}} \cdot 0.0001 \\ \alpha_{\text{start}} \cdot \left(1 - \frac{N_t}{N_e \cdot N + 1}\right), & \text{otherwise} \end{cases},$$

    where N_t is the number of currently processed words, N is the total number of words in the training text and N_e is the number of epochs to run.

4. **Find the closest words**. Reuse your nearest-neighbor implementation from the previous task for implementing the `find_nearest` function.

You can test your code by simply running:

```
python w2v.py
```

Compare your word2vec embeddings with your Random Indexing embeddings. Which ones would you use in practice?

3. (Word embeddings as features) In this problem we will explore how well word embeddings can be used as features for the familiar task of Named Entity Recognition (NER), which we encountered already in assignment 2. Recall that NER is the binary classification problem of predicting whether a given word is a named entity or not. However, instead of using hand-crafted features, we will now use word embeddings directly and see if we get any performance boost.

In order to pass this problem you will need to train a classifier capable of discriminating between classes in a non-trivial way (i.e. not classifying all data points as a names, or all data points as non-names). Note that the performance for such a classifier might not necessarily be better than the most frequent class baseline. The plan for this problem looks roughly as follows:

1. Download pre-trained word2vec embeddings for English (for instance, download the ones trained for CoNLL 2017 Shared Task).
2. Train Random Indexing word embeddings on corpus that is both larger than Harry Potter books and more general-purpose. For instance, NewsCrawl 2010 from WMT 2014 Shared Task that could be downloaded here.
3. Re-use your binary logistic regression code from the assignment 2 to train a classifier that uses word2vec embeddings.
4. Train another classifier using your freshly trained RI embeddings.
5. Experiment with hyper-parameters to find the best classifiers you can in 3 and 4.

Here are some hints to help you out.

- Word vector files can be quite large (a couple of GB), so you might find yourself out of RAM pretty quickly. To aid you, we provide a modified `NER.py` that uses Python's memory mapping to deal with the problem.
- Random Indexing vectors can grow quite large in values due to the nature of the algorithm, so you might end up with overflows quite quickly. One way out of this is to normalize the vectors.
- The convergence criterion you've used in the assignment 2 might not work simply because the gradient vector has now more dimensions, so it might take quite long for all partial derivatives (or their sum) to be under the threshold. Instead you might want to either train for a fixed number of iterations or employ early stopping.
- The dataset is pretty imbalanced by nature, i.e., there will always be much more non-names than names. One way to counteract this imbalance, one could introduce weights to the cross-entropy loss, i.e.:

$$L = \frac{1}{N} \sum_{i=1}^{N} w_p y_i \log h(x_i) + w_n (1 - y_i) \log(1 - h(x_i))$$

  where x_i is a feature vector and y_i is a label for the word i, $h(\cdot)$ is the logistic regression model and w_p (w_n) is the weight for the positive (negative) instances. The idea is to set the weights such that the cumulative contributions of both positive and negative datapoints are equal. A good starting point is to set $w_p = \frac{N_n}{N}$ and $w_n = \frac{N_p}{N}$, where N_n (N_p) is the number of negative (positive) datapoints. **Remember to change your gradient computations accordingly**.

Optional part

4. (GloVe) A more recent approach to word embeddings is the *Global Word Vectors* (GloVe) approach by Pennington, Socher and Manning (2014). For details on the GloVe training algorithm, see the lecture slides and/or the original paper (Pennington et al., 2014). Your task in this problem is to complete the provided code skeleton so that the program correctly implements the GloVe algorithm, and then train GloVe word embeddings on the Harry Potter corpus. We require only that you train on the first Harry Potter book, but the results will be much better if you train on all 7 books.

   The program is organized in two files: `Glove.py` that does the training, and `GloveTester.py`, which allows you to check the correctness of your implementation by querying the nearest neighbors of the same words as in the earlier problems. The training program saves the obtained vectors every 1,000,000 iterations, so you can test your current vectors while the training is running. (The neighbors will not necessarily be exactly the same as those computed by Random Indexing, but they should make sense.)

   Here are some hints to help you out:

   1. Use stochastic gradient descent. In every iteration, choose words i, j randomly, compute the gradient of the loss function w.r.t. w_i and $\widetilde{w_j}$, and update w_i and $\widetilde{w_j}$ based on those gradients.
   2. In order to get good results without waiting for an eternity, choose the word i in (1) according to how common it is. That is, if a word co-appears with many other words according to the X matrix, it should have a higher probability of being chosen for updating. The `random.choices` library method is useful for this purpose.
   3. As a convergence criterion, check whether the overall loss has increased. You may compute the loss, say, every 100,000 iterations. Don't break immediately if the loss increases, but rather use early stopping with patience = 5.
   4. Start with learning rate = 0.05, and decrease it slightly every 1,000,000 iterations.

References

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 3111–3119.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.