Applied Machine Learning Exercise #2

Gal Lavee
May 29, 2018

In this exercise we will implement a word2vec word embedding model. Code can be written in the programming language of your choice, but no external libraries are allowed, with the sole exception of libraries that perform vector and matrix calculation (e.g. numpy in python or Math.Net Numerics for C#). You may also using plotting libraries or other tools for generating the required plots. You can also use randomization functions (such as *random.multivariate_normal* , *random.choice* in python). Update 5/29/2018: Changes marked in orange

Part 1. The Data

Download and unzip the Stanford Treebank data[1]. The file *README.txt* describes the different files included. For our purposes we will use only *dataSetSentences.txt* and *datasetSplit.txt*.

In this part you are required to :

1. implement a class to parse the sentence assignments to training and test sets. The constructor of the class should take as an argument the path to*datasetSplit.txt*.
2. implement a class to hold the training/test data. The constructor of the class should receive the path to *dataSetSentences.txt* and an instance of the class implemented above (to indicate assignment of sentences to train/test). Each sentence should be stored as an (ordered) list of tokens which are the words(split on whitespace) of the sentence after the following preprocessing:
   (a) lowercase
   (b) removal of non-ascii characters
   (c) removal of non-alphanumeric characters
   (d) removal of words with less than 3 characters

Part 2. The Model

Recall from lecture that under the skip-gram model we are trying to maximize the probability of "context words" given the "input word". Consider the sentence *Gal Loves IDC*. Using a context window of size 1 (on each side of the input), would yield the following input and context word combinations:

$$\underbrace{\text{Gal}}_{input}\quad \underbrace{\text{Loves}}_{context}\quad \text{IDC}$$
$$\underbrace{\text{Gal}}_{context}\quad \underbrace{\text{Loves}}_{input}\quad \underbrace{\text{IDC}}_{context}$$
$$\text{Gal}\quad \underbrace{\text{Loves}}_{context}\quad \underbrace{\text{IDC}}_{input}$$

[1] http://nlp.stanford.edu/~socherr/stanfordSentimentTreebank.zip

In the negative-sampling variant of *word2vec* the (log) probability of seeing a context word in the dataset, denoted w_c below, given an input word, denoted w_i, is given by the expression:

$$\log P\left(w_c \mid w_i\right) = \log \sigma\left(\mathbf{u}_i^\top \mathbf{v}_c\right) + \frac{1}{K} \sum_{k=1}^{K} \log\left(1 - \sigma\left(\mathbf{u}_i^\top \mathbf{v}_k\right)\right) \tag{1}$$

This implies all gradient involving the negative samples will now involve the $1/K$ factor.

where $\mathbf{u}_i \in \mathbb{R}^d$ and $\mathbf{v}_i \in \mathbb{R}^d$ are the 'target' and 'context' representation of each word in our vocabulary. These are our model parameters, collectively denoted Θ.

$$\Theta = \left\{ \{\mathbf{u}_i\}_{i=1}^{|V|}, \{\mathbf{v}_i\}_{i=1}^{|V|} \right\}$$

Equation 1 also assumes that K 'non-context' words, $w_1, \ldots, w_K$ are sampled from some distribution over the vocabulary, V.
In this part you are required to :

1. implement a class representing the model hyperparameters which include: the size of the context window, the size of the vector representation of each word, the number of "non-context" negative words, the number of iterations between reductions of the learning rate, the choice of noise distribution, and random seed governing randomness in initialization(to allow reproducibility)

2. implement a class that represents your model parameters. This class should take as input an instance of the hyperparameters class

3. implement a member function *Init*, that takes the training data as input, to initialize the parameters as follows:

   (a) Create 2 vectors (one $\mathbf{v}$ one $\mathbf{u}$) for each word in the training data

   (b) Sample each vector from a multivariate Gaussian with mean $\mathbf{0}$ (vector) and Covariance $0.01 \cdot \mathbf{I}_D$ (the Identity matrix)

   (c) Normalize the random vector to be unit length (using the L2 norm)

4. implement a function to sample words from the noise distribution:

   $$P_n(w) \propto U(w)^\alpha$$

   where $U(w)$ is the *unigram distribution* and $0 \leq \alpha \leq 1$ is a parameter. The unigram distribution should be calculated based on the training data.

5. implement a function to sample K random words from the vocabulary using the noise distribution specified in the hyperparameters

6. implement a function to return the log probability of context word i given input word j and K negative samples.

Part 3. The Algorithm

In this section we will use *stochastic gradient descent*(SGD) (actually ascent)to find the setting of model parameters, Θ which maximize the (log) Likelihood:

$$L(\Theta) = \sum_{(w_c, w_i) \in \mathcal{D}} \log P(w_c \mid w_i) \tag{2}$$

where $\mathcal{D}$ is the set of all context/input word pairs in our training data.

Rather than use the entire dataset to compute the gradient we will use a small subset of context/input pairs, selected uniformly at random, called a *mini-batch*, to perform the optimization. Each iteration of SGD will thus use the following update rule:

$$\Theta_{new} = \Theta_{old} + \eta \left(\sum_{(w_c, w_i) \in \mathcal{D}_{mb}} \nabla_{\Theta} \left[\log P(w_c \mid w_i) \right] (\Theta_{old}) \right) \tag{3}$$

where η is a hyperparameter called the learning rate, and $\mathcal{D}_{mb}$ is a mini-batch of context/input pairs.
In this part you are required to :

1. Derive the gradient that appears in Equation 3 with respect to *all* $\mathbf{u}_i$ and $\mathbf{v}_i$

2. using your derivation above implement a function that computes the gradient updates given a particular context/input pair and K negative samples.

3. implement a function that samples a context/input from the training data. This function should take the context window length, C as a parameter and return a list of context/input word pairs (one for each context word in the context window). Let us adopt the convention that context windows are always symmetric about the input word and we take C context words on either side of the input word. That is, each context window will have $2C$ context words (you may choose how to deal with words on the edges of the sentence as you wish).

4. implement a function that uses the above function to create a mini-batch of context/input pairs.

5. implement a function that updates the model parameters using the SGD rule in equation 3 given a mini-batch

6. implement a class encapsulating all algorithm hyperparameters including learning rate and mini-batch size

7. implement a function called *LearnParamsUsingSGD* which takes a training set and hyperparams as input and runs the SGD update on a randomly sampled mini-batch at each iteration for a pre-specified number of iterations. After each update the parameters should be renormalized so that all vectors are unit length (in terms of the L_2 norm). After a fixed number of iterations(specified in the hyperparameters) has elapsed the learning rate should be reduced by 50%.

Part 4. Machine Learning Diagnostics

Since we are maximizing the objective in Equation 2, one might imagine computing the value of this objective every few iterations to see if our algorithm is learning. However, this calculation is heavy as there are many elements in the summation. Instead, we can evaluate the 'mini-batch likelihood' at each iteration as a surrogate:

$$L(\Theta) = \sum_{(w_c, w_i) \in \mathcal{D}_{mb}} \log P(w_c \mid w_i) \tag{4}$$

because each mini-batch is slightly different, these quantities will not be strictly increasing at every iteration. However, their magnitude should show a trend to larger (negative) numbers.

In this part you are required to:

1. Implement a function that computes the log-likelihood given in Equation 2 which takes the model hyperparameters as input
2. Implement a function that computes the 'mini-batch likelihood' given in Equation 4
3. Modify the function *LearnParamsUsingSGD* from the previous part to print the mini-batch likelihood at each iteration (after the parameter update)
4. Modify the function *LearnParamsUsingSGD* to take a second dataset (i.e. a test set). Every X iterations, where X is a hyper-parameter the function should save the iteration number and the value of the mini-batch log-likelihood on the train set and the full log-likelihood on the test sets, given the current settings of the parameters. In order to make the numbers on the same scale report the *mean* log-likelihood across all examples considered.

Part 5. Evaluation

There are few neat ways to evaluate word2vec models beyond measuring log-likelihood. Some ideas include:

- predicting the most likely context given an input word :

$$\hat{w}_c = \arg\max_{w_c} P(w_c \mid w_i)$$

- predicting the input word given a particular string of words, $w_1, \ldots w_N$:

$$\hat{w}_i = \arg\max_{w_i} P(w_1, \ldots, w_{i-C}, \ldots w_i, \ldots w_{i+C}, \ldots w_N) = \arg\max_{w_i} \prod_{c=-C}^{C} P(w_{i+c} \mid w_i)$$

- Solving 'SAT analogy questions', for example "man is to woman as king is to __?" using linear relations on the word embeddings in the model as follows:

$$\text{solution to analogy} = \arg\max_{i} \mathbf{u}_i^\top (\mathbf{u}_{man} - \mathbf{u}_{woman} + \mathbf{u}_{king})$$

In this part you are required to:

1. Implement a function that takes in a model and an input word and outputs the 10 most likely contexts.

2. Implement a function that takes in a model and a set of context words and outputs the 10 most likely inputs
3. Implement a function that takes a list of input words and visualizes the words on a scatter plot using the first 2 elements of each word's embedding vector.
4. implement an 'analogy solver' function which takes the first three parts of an analogy and outputs the top 10 results (along with their score) for the last part

Part 6. The Main Flow

In this part we will put all the pieces together. Implement the main flow of your program to do the following:

1. Read in data and split into train and test
2. Set hyperparameters based on input arguments/configuration file
3. Learn model using algorithm specified by input arguments/configuration file (should output train and test performance to files)
4. Output the hyperparameters and final (mean) log-likelihoods to a file, the file should also include the amount of time needed for training.

Part 7. Diagnostics and Analysis

In this part you will use the code you implemented above to create several plots (marked by green Deliverable tag) that you will turn in along with your code

In order to make sure our algorithm is implemented correctly we want to see that the training log-likelihood is being increased over time in our algorithm

Deliverable 1. Plot the log-likelihood train and test as a function of training iteration. Train and test plots should appear in the same figure, clearly marked. You may use whatever configuration of hyperparameters you wish for this plot, but make sure you specify the choice.

Hyperparameters can sometimes have a great effect on generalization performance. We will consider the effect of the embedding size and other hyperparams.

Deliverable 2. Set the hyper params as follows:

1. Learning Rate = 0.01 – 1.0
2. Num iterations = 20000
3. Maximum context window size = 5
4. mini-batch size = 50
5. noise distribution = Uniform ($\alpha = 0$) (or some other small value of α e.g. 0.01)
6. number of negative samples per context/input pair (denoted K above)=10

Vary the size of the word embedding ,d from 10 to 300 in 5 evenly spaced intervals. In two separate plots, plot both training time and train and test (mean) log-likelihood as a function of d. All hyper-parameter configurations of the algorithm should be clearly specified.

Deliverable 3. Repeat the setup above, but this time fixing d (to your choice) and varying one of {learning rate , mini-batch size, noise distribution} (again your choice). As before, generate two plots one for training time and one for train and test log-likelihood. Clearly specify all your choices.

Deliverable 4. Consider the following words: {*good*, *bad*, *lame*, *cool*, *exciting*}

1. Print the top 10 *context* words according to the model when each of the above is considered to be an input word
2. Learn a model with $d = 2$, show a scatter plot visualizing the embedding of the above words. Add additional words of your choice to the plot. Try using both the input and the context embeddings. Clearly specify all hyperparameters used.

Deliverable 5. Consider the following incomplete sentences:

1. *The movie was surprisingly __.*
2. __ *was really disappointing.*
3. *Knowing that she __ was the best part.*

Learn a model with your choice of hyperparameters (you may use one from previous sections). Use the model to complete the sentences (print 10 best completion) by considering the non-blank words as context. Clearly specify all hyperparameters used.

Deliverable 6. Consider the following analogy questions:

1. *man is to woman as men is to* __
2. *good is to great as bad is to* __
3. *warm is to cold as summer is to* __

Learn a model with your choice of hyperparameters (you may use one from previous sections). Use the model to answer the analogy questions, using the linear approach specified above (print 10 best completions). What happens when you use context embeddings instead of input embeddings? Clearly specify all hyperparameters used.

Deliverable 7. Your Code. Be prepared to explain where and how you implemented each of parts 1-6 above.