Practice 6

FastText

FastText was introduced by back in 2017. The idea behind FastText is very similar to Word2Vec. However, there was still one thing that methods like Word2Vec and GloVe lacked.

If you've been paying attention, you must have noticed one thing that Word2Vec and GloVe have in common – how we download a pre-trained model and perform a lookup operation to fetch the required word embeddings. Even though both of these models have been trained on billions of words, that still means our vocabulary is limited.

So what makes FastText different [https://arxiv.org/pdf/1712.09405.pdf]?

1. **CBOW(skipgram) + negative sampling**. For the model of vector representations of words, CBOW with negative sampling is used. Negative sampling is a way to create negative examples for training a vector model. During training pairs of words are shown that are not neighbors in context. For each positive example (when the words in the text are next to each other, for example, "fluffy cat"), we select several negative ones ("fluffy iron", "fluffy radio signal", "fluffy escape"). In total, from 3 to 20 random words are selected. Such a random selection of several examples does not require much computer time and allows you to speed up FastText.

2. **Subword model.** CBOW (skipgram) ignores word structure, but some languages have compound words. Therefore, a subword model was added to the main model. A subword model is a representation of a word through chains of characters (n-grams) with n from 3 to 6 characters from the beginning to the end of the word, plus the entire word itself. For example, the word lock with n = 3 will be represented by 3-grams <re, rea, ead, adi, din, ing, ng> and the sequence <reading>. This approach allows you to work with those words that the model has not seen before.

For the word '*reading'*, character n-grams of length 3-6 would be generated for this word in the following manner (angular brackets denote the beginning and the end of the word.):

Word	n-gram	combinations
reading	3	<re, rea, ead, adi, din, ing, ng>
reading	4	<rea, read, eadi, adin, ding, ing>
reading	5	<read, readi, eadin, ading, ding >
reading	6	<readi, readin, eading, ading>

Each word is decomposed into its character n-grams N and each n-gram n is represented by a vector x_n. The word vector is then simply the sum of both representations, i.e.:

$$v_w + \frac{1}{|N|}\sum_{n \in N} x_n.$$

In practice, the set of n-grams *N* is restricted to the n-grams with 3 to 6 characters. Storing all of these additional vectors is memory demanding. This simple model allows sharing the representations across words, thus allowing to learn reliable representation for rare words.

3. The features obtained by splitting into n-grams have a huge dimension (i.e. a huge heavy table is obtained for the text). This can slow down the performance of a model trained on these features. To fix the dimensions of features, **feature hashing** is used (a special procedure that allows you to encode objects of different sizes using character strings of the same length). The **Fowler-Noll-Vo** [http://www.isthe.com/chongo/tech/comp/fnv/] hashing function that maps n-grams to integers in 1 to *K*, $K = 2 \times 10^6$ is used. Features get hash indexes, which helps to read them faster. Ultimately, a word is represented by its index in the word dictionary and the set of hashed n-grams it contains. Although this hashing approach results in collisions, it helps control the vocabulary size to a great extent.

4. When the FastText word representation is used for text classification and analysis, the greater the number of classes, the longer the running time of the linear model. To optimize the classifier, a hierarchical softmax based on the Huffman coding algorithm is used [https://aclanthology.org/E17-2068.pdf].

You can use FastText either by the native library [https://fasttext.cc/docs/en/support.html] or by Gensim.

FastText with Gensim

Use FastText with genism library:

```
from gensim.models import FastText
# some example sentences
from gensim.test.utils import common_texts

print(common_texts[8])          # output the example of text
print(len(common_texts))        # output the number of texts

tokenised_sentences = common_texts[:]
tokenised_sentences
```

To create a FastText object:

```
model = FastText(window=3, min_count=1)  # instantiate
```

The constructor has the following default parameters:

```
class gensim.models.fasttext.FastText(sentences=None, corpus_file=None, sg=0,
hs=0, size=100, alpha=0.025, window=5, min_count=5, max_vocab_size=None,
word_ngrams=1, sample=0.001, seed=1, workers=3, min_alpha=0.0001, negative=5,
```

```
ns_exponent=0.75, cbow_mean=1, hashfxn=<built-in function hash>, iter=5,
null_word=0, min_n=3, max_n=6, sorted_vocab=1, bucket=2000000, trim_rule=None,
batch_words=10000, callbacks=())
```

Constructor contains the following parameters:

- **sentences** - iterable of tokenized texts
- **size** - dimensionality of learned embeddings(Default 100)
- **window** - distance from target word considered same-context (Default 5)
- **min_count** - the minimum number of times the word or word-piece must occur to be included in our vocabulary(Default 5)
- **model**: Training architecture. Allowed values: `cbow`, `skipgram` (Default `cbow`)
- **alpha**: Initial learning rate (Default 0.025)
- **loss**: Training objective. Allowed values: `ns`, `hs`, `softmax` (Default `ns`)
- **sample**: Threshold for downsampling higher-frequency words (Default 0.001)
- **negative**: Number of negative words to sample, for `ns` (Default 5)
- **iter**: Number of epochs (Default 5)
- **sorted_vocab**: Sort vocab by descending frequency (Default 1)
- **threads**: Number of threads to use (Default 12)

Additional parameters

- **min_n**: min length of char ngrams (Default 3)
- **max_n**: max length of char ngrams (Default 6)
- **bucket**: number of buckets used for hashing ngrams (Default 2000000)

To build the vocabulary and train the model:

```
model.build_vocab(tokenised_sentences)

model.train(sentences, total_examples=len(sentences),
epochs=10)  # train
```

Once you have a model, you can access its keyed vectors via the *model.wv* attributes. The keyed vectors instance is quite powerful: it can perform a wide range of NLP tasks.

You can also pass all the above parameters to the constructor to do everything in a single line:

```
model2 = FastText(vector_size=4, window=3, min_count=1,
sentences=common_texts, epochs=10)
```

Task 1: Explain the passed parameters to the model.

The two models above are instantiated differently, but behave identically. For example, we can compare the embeddings they've calculated for the word "computer":

```
import numpy as np
np.allclose(model.wv['computer'], model2.wv['computer'])
```

True

To get a word vector:

```
model.wv[' you ']
```

Output similar words vectors:

```
model.most_similar("happy")
```

Check if the word is in vocabulary:

```
print('night' in model.wv.vocab)
```

Note that the word vector lookup operation only works if at least one of the component character n-grams is present in the training corpus.

Similarity operations work the same way as word2vec. Out-of-vocabulary words can also be used, provided they have at least one character n-gram present in the training data.

```
print("nights" in model.wv.vocab)
print("night" in model.wv.vocab)
model.similarity("night", "nights")
```

To load and save model:

```
#  Model saving and loading
model.save(fname)
model = FastText.load(fname)
```

Once loaded, such models behave identically to those created from scratch. For example, you can continue training the loaded model:

```
old_vector = np.copy(model.wv['computation'])  # Grab the existing vector

new_sentences = [
    ['computer', 'aided', 'design'],
    ['computer', 'science'],
    ['computational', 'complexity'],
    ['military', 'supercomputer'],
    ['central', 'processing', 'unit'],
    ['onboard', 'car', 'computer'],
]
model.build_vocab(new_sentences, update=True)  # Update the vocabulary
model.train(new_sentences, total_examples=len(new_sentences),
epochs=model.epochs)

new_vector = model.wv['computation']

np.allclose(old_vector, new_vector, atol=1e-4)  # Vector has changed, model
has learnt something
```

Important! Be sure to call the build_vocab() method with *update=True* before the train() method when continuing training. Without this call, previously unseen terms will not be added to the vocabulary.

Task 2: Download FastText embeddings from here: https://fasttext.cc/docs/en/english-vectors.html

Task 3: Define a function `similar_word(vector)`, that finds 10 nearest words with an Euclidian metrics. Output 10 nearest words to word `'holiday'`.

Task 3: Update the model on 3 first texts of true-fake news. Do not forget to tokenize texts! Check, if vectors were changed.

Task 4: Visualize 15 clothest words to the word “president” by Eucleadian metrics. Compare the results with word2vec and GloVe.

Task 5: Calculate the result of: `model.most_similar(positive=['king', 'queen'], negative=['man'])`. Compare the results with word2vec and GloVe.

Questions to Practice 6:

1. What is **negative sampling**?
2. How does **Subword model** for **FastText** algorithms is implemented?
3. How input vector to the network is calculated?
4. Why do we need to use hash function?
5. What for is Huffman coding algorithm used during text classification?