SkipGram rocks!

As suggested by the given template, we will construct a class `SkipGram` that encapsulates all the hard work. The OOP frameword will also prove useful in efficiently running experiments.

Table of content:

---

1. Preprocessing is done with `__init__` method

As the resulting `SkipGram` will be trained on a unique corpus, many tasks specific to the corpus will be done once (*) within our `SkipGram`. In consequence, our `__init__` method will serve four goals:

1.1. Text preprocessing

We use a white space stemmer as a baseline in this work. Additionally, we remove punctuations as it oftentimes leads to duplicate of words ('bye' and 'bye.'). We choose to use list comprehension motivated by this Quora question. This can be optimized using C based look-up tables. We also remove numbers.

```
def preprocess_line(line):
    lower = line.lower()
    punctuation = ''.join(c for c in lower if c not in string.punctuation)
    digits = ''.join([i for i in punctuation if not i.isdigit()])
    clean_line = digits.split()
    return(clean_line)
```

1.2. Hyperparameters

We set the hyperparameters specific to the model (`window_size`, `embedding_dimension`...) as attributes and we add all remaining hyperparameters (`epochs`, `stepsize`...) for to centralize them altogether.

```
# ATTRIBUTES
## Basic attributes
self.window_size = window_size
self.min_count = min_count
self.negative_rate = negative_rate
self.embedding_dimension = embedding_dimension
## Custom attributes
self.epochs = epochs
self.stepsize = stepsize
self.dynamic_window = dynamic_window
self.subsampling = None or 1e-5
```

Edit: We added several attributes corresponding to the use of techniques such as subsampling: - `dynamic_window` as an attribute of the model. - `subsampling` set to either the threshold or to `None`, if subsampling should not be applied.

1.3. Vocabulary

The vocabulary is a key component of the SkipGram model. Its main goal is to construct an integer indexing of the words (e.g. two dictionaries `self.word2idx` and `self.idx2word`). For convenience purposes, the indexes are sorted by the number of occurences. We use `np.unique` with `return_counts = True` to obtain the `unique` words of the corpus and their respective `count`. We save the number of occurences in a dictionary `self.wordcount` which will be useful for subsampling.

```
# CREATE VOCABULARY
## Count words and create a sorted dictionary
unique, count = np.unique(np.array([x for sentence in sentences for x in sentence]), return_
idx = np.where(count > min_count)
unique, count = unique[idx], count[idx]
## Word count
self.word2count = dict(zip(unique, count))
```

```
self.word2count = dict(sorted(self.word2count.items(), key=lambda kv: kv[1]))

## Retrieve vocabulary size after min count
self.vocab_size = len(self.word2count)

## Create mapping between words and index
self.idx2word = dict(zip(np.arange(0, self.vocab_size), self.word2count.keys()))
self.word2idx = {v:k for k, v in self.idx2word.items()}
```

1.4. Corpus encoding

We convert the list of sentences by replacing each word by its index from `word2idx`. List comprehension leads to more efficient computations of words contexts and is lighter in terms of storage.

```
# Encode corpus
self.corpus_idx = [[self.word2idx[w] for w in sent] for sent in sentences]
```

1.5. Negative sampling distribution

Prepare Negative sampling upstram (because *) by computing its word distribution `word_distribution` and the corresponding `normalization_cte` (see Section 3 for more details). Additionally, as these solely rely on the occurences of each word (see `word2count`), it makes sense to integrate it in the `__init__`.

```
# Create distribution
self.normalization_cte = sum([occurence**0.75 for occurence in self.word2count.values()])
self.word_distribution = {k:v**0.75/self.normalization_cte for k, v in self.word2count.items
```

This distribution is only sampled during the training phase, in `run_batch`!

2. The full `train` procedure

2.0. The gradients in python

Learning is done with gradient descent (see `report.pdf` (section I) for the mathematical derivation). I python, we can vectorize the gradient computations with:

```
def gradients_custom(target, prediction, current_word, context_words):
    # Compute error
    error = prediction - target

    # Exact gradients
    grad_w0 = np.sum(error*context_words, axis = 1)
```

```
    grad_w1 = np.outer(current_word, error)

    return(grad_w0, grad_w1)
```

2.1. An optimized learning procedure

The `train` method incorporates the three tasks of SkipGram coupled with Negative sampling:

1. Compute the in-context with sliding windows around the centered word `w`
2. Compute the out-contexts with negative sampling
3. Run batch with target word (label 1) and its negative samples (labels 0)

Let's initialize uniformly the weights:

```
w0 = np.random.uniform(-1, 1, size = (self.vocab_size, self.embedding_dimension))
w1 = np.random.uniform(-1, 1, size = (self.embedding_dimension, self.vocab_size))
```

At first, and because all these tasks require going over all the words of the corpus once, we used nested `for` loops, we wrote something along the lines of:

```
# Each epochs go over all sentences of the corpus.
for epoch in range(0, self.epochs):
    # The current sentence of the corpus.
    for s, encoded_sentence in enumerate(self.corpus_idx):
        # Retrieve the `context` words for each `w`
        for w, token in enumerate(encoded_sentence):
            # For each context word and its drawn negative samples
            context = retrieve_context(w, self.window_size, encoded_sentence)
            for context_word in context:
                # Batch backprop of the neural network
                w0, w1, ... = run_batch(w0, w1, ...)
```

This approach proved to be highly inefficient because except for running the batch in itself, the other tasks like context retrieval and drawing negative samples (see section 3.4.) can be done `ONCE` efficiently. For the contexts, we use python `map` operator instead of `for` loop.

```
# Context creation
contexts_corpus = map(lambda x: contexts_sentence(x, self.window_size, self.dynamic_window),
# Flatten nested list
contexts_corpus = [item for sublist in contexts_corpus for item in sublist]
```

Altogether we obtain the optimized learning procedure:

```
## Repeat training on whole corpus self.epochs times
for epoch in range(0, self.epochs):
    ## For every word in the corpus
```

```
    for i, current in enumerate(current_words):
        w0, w1, ... = run_batch(self, current, contexts_corpus[i], w0, w1, ...)
```

This version is inefficient for many reasons (see 3.4.) but allowed us to make `SkipGram` work in the first place.

2.2. Retrieve the contexts

The basic idea behind computing `context_word` is to go over all the words in the corpus (much like one would read a corpus) and see which word is near the current word within a given window. At a given word in this "reading", we retrieve its nearby words in python using:

```
def retrieve_context(w, window_size, encoded_sentence, dynamic_window = False):

    # Change to dynamic window
    if dynamic_window: window_size = np.random.randint(1, window_size+1)

    # List comprehension to retrieve the words
    context = encoded_sentence[max(w - window_size, 0) : w] + \
              encoded_sentence[w + 1: min(w + window_size + 1, len(encoded_sentence))]

    return(context)
```

2.3. How to construct and learn a batch? - Stucture OK

While the the weights `w0` and `w1` are initialized prior the epochs loop, the whole embedding learning can be boiled down to a batch. Word2vec thereby learns by running (lots of) batches, with each batch consisting of the `current word`, `contexts_word` and corresponding `negative_samples`.

```
def run_batch(sg, current, contexts_word, negative_samples, w0, w1, epoch_error, sgd_count):
    # PREPARE BATCHE
    # FORWARD PASS
    # BACKWARD PASS
    return(w0, w1, epoch_error, sgd_count)
```

Preparing the batch: All we need are the in and out contexts (see `negative_samples` and `contexts_word`), because we know their respective `labels`. The `negative_samples` are drawn at each epoch (see 3.4.) and passed through

```
batch_contexts = contexts_word + list(negative_samples)
target = np.array([1 for ctxt in contexts_word] + [0 for i in negative_samples])
```

Forward pass: We retrieve the `w0` of the current word and the embeddings `w1` of the context words. We take the sigmoid of their dot product.

```
# Get embeddings and weights
current_word = w0[current,:]
context_words = w1[:, batch_contexts]
# Probabilities
output = expit(np.dot(current_word, context_words))
```

Backward pass: We compute the gradients as explained in section 3.0. and update the weights.

```
# Gradient computations
gradient_loss = gradients_custom(target, output, current_word, context_words)
# Update
w0[current,:] -= sg.stepsize * gradient_loss[0]
w1[:, batch_contexts] -= sg.stepsize * gradient_loss[1]
```

Implementation note: We track the progress with `epoch_error` and `sgd_count`.

```
# Keep track of the error (mean probabilities error)
epoch_error += np.sum(np.abs(output - target))
sgd_count += target.shape[0]
```

2.4. Negative sampling in practice

The Negative sampling works by generating k negative samples for each `context_word`. The definition of the distribution used for sampling has been done in section 2.5. As for sampling, it is standard to proceed with `np.random.choice`. At first, they were drawn within each batch. This slow down the learning process immensely. Therefore, we draw all negative samples at once before the first epoch (with `replace = True`). In python, this translates to:

```
negative_samples = np.random.choice(list(self.word2idx.values()),
                                    size = (len(contexts_corpus), self.negative_rate * self.
                                    p=list(self.word_distribution.values()),
                                    replace = True)
```

3. Improve Word2Vec

3.1. Dynamic window size

As a dynamic window size arguably leads to better results, we integrate this to our code using the attribute `self.dynamic_window = True` to the `__init__` method. As for the actual computations, we use randint function from numpy. Altogether, this gives

```
if self.dynamic_window: win = np.random.randint(1, self.window_size + 1)
else: win = self.window_size
```

3.2. Effect of subsampling and rare-word pruning

Min count: Trying to learn contexts of rare words in the corpus is problematic for SkipGram, as there is not enough samples to train properly. And because there are less contexts to learn, this boosts the overall training speed. We retain words with at least `self.min_count = 5` occurences.

Subsampling: Implemented as is, the frequent words such as 'the', 'is' undermines the ability of SkipGram to learn great embeddings. To tackle this issue, frequent words are down-sampled. The motivation behind this variation is that frequently occuring words hold less discriminative power. The underlying motivation is the effect of increasing both the effective window size and its quality for certain words. According to Mikolov et al. (2013), sub-sampling of frequent words improves the quality of the resulting embedding on some benchmarks.

```
frequencies = {word: count/self.total_words for word, count in self.word2count.items()}
drop_probability = {word: (frequencies[word] - subsampling)/frequencies[word] - np.sqrt(subs
self.train_words = {k:v for k, v in drop_probability.items() if (1 - v) > random.random()}
self.corpus_idx = [[self.word2idx[w] for w in sent if w in self.train_words.keys()] for sent
```

Implementation note: Importantly, these words are removed from the text before generating the contexts (in `__init__`).

4. Experiments

4.1. Evaluation

We evaluate the embeddings quality using SimLex 999. The results are in report.pdf and evaluation.ipynb

4.2. Post-processing embeddings

See `pca_w2v` in code/evaluation.py and the PCA section in report.pdf.

5. Future work

5.1. Character n-grams

This is motivated by these answers. See python implementation. This one seems more nice and efficient though.

5.2. Algebraic properties of the embeddings

Word embeddings are said to have remarkable properties, such as:

$$v_{queen} = v_{king} + (v_{man} - v_{woman})$$

These geometric structures are hard to understand to date. There are definitely some experiments that should be done on this topic.