Build vocabulary and data iterator

In this notebook we are going to create the vocabulary object that will be responsible for:

- Creating dataset's vocabulary.
- Filtering dataset in terms of the rare words occurrence and sentences lengths.
- Mapping words to their numerical representation (word2index) and reverse (index2word).
- Enabling the use of pre-trained word vectors.

The second object to create is a data iterator whose task will be:

- Sorting dataset examples.
- Generating batches.
- Sequence padding.
- Enabling BatchIterator instance to iterate through all batches.

Let's begin with importing all necessary libraries.

In [19]:

```
import pandas as pd
import numpy as np
import re
import torch
from collections import defaultdict, Counter
from pprint import pprint
import warnings
warnings.filterwarnings('ignore')
```

Now we are going to build the vocabulary class that includes all the features mentioned at the beginning of this notebook. We want our class to enable to use of pre-trained vectors and construct the weights matrix. To be able to perform that task, we have to supply the vocabulary model with a set of pre-trained vectors.

Glove vectors can be downloaded from the following website: https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/)
Fasttext word vectors can be found under the link: https://fasttext.cc/docs/en/english-vectors.html (https://fasttext.cc/docs/en/english-vectors.html)

In [20]:

```
class Vocab:

    """The Vocab class is responsible for:
    Creating dataset's vocabulary.
```

```
Filtering dataset in terms of the rare words occurrence and senter
Mapping words to their numerical representation (word2index) and r
Enabling the use of pre-trained word vectors.


Parameters
----------
dataset : pandas.DataFrame or numpy.ndarray
    Pandas or numpy dataset containing in the first column input s
    variable as last column.
target_col: int, optional (default=None)
    Column index refering to targets strings to process.
word2index: dict, optional (default=None)
    Specify the word2index mapping.
sos_token: str, optional (default='<SOS>')
    Start of sentence token.
eos_token: str, optional (default='<EOS>')
    End of sentence token.
unk_token: str, optional (default='<UNK>')
    Token that represents unknown words.
pad_token: str, optional (default='<PAD>')
    Token that represents padding.
min_word_count: float, optional (default=5)
    Specify the minimum word count threshold to include a word in
    If min_word_count <= 1 then keep all words whose count is grea
    of the count distribution.
max_vocab_size: int, optional (default=None)
    Maximum size of the vocabulary.
max_seq_len: float, optional (default=0.8)
    Specify the maximum length of the sequence in the dataset, if
    the maximum length to value corresponding to quantile=max_seq_
    sequences whose lengths are greater than max_seq_len.
use_pretrained_vectors: boolean, optional (default=False)
    Whether to use pre-trained Glove vectors.
glove_path: str, optional (default='Glove/')
    Path to the directory that contains files with the Glove word
glove_name: str, optional (default='glove.6B.100d.txt')
    Name of the Glove word vectors file. Available pretrained vect
    glove.6B.50d.txt
    glove.6B.100d.txt
    glove.6B.200d.txt
    glove.6B.300d.txt
    glove.twitter.27B.50d.txt
    To use different word vectors, load their file to the vectors
weights_file_name: str, optional (default='Glove/weights.npy')
    The path and the name of the numpy file to which save weights

Raises
-------
ValueError('Use min_word_count or max_vocab_size, not both!')
```

```
        If both: min_word_count and max_vocab_size are provided.
    FileNotFoundError
        If the glove file doesn't exists in the given directory.

    """


    def __init__(self, dataset, target_col=None, word2index=None, sos_
             pad_token='<PAD>', min_word_count=5, max_vocab_size=None,
             use_pretrained_vectors=False, glove_path='glove/', glove_
             weights_file_name='glove/weights.npy'):

        # Convert pandas dataframe to numpy.ndarray
        if isinstance(dataset, pd.DataFrame):
            dataset = dataset.to_numpy()

        self.dataset = dataset
        self.target_col = target_col

        if self.target_col:
            self.y_lengths = []

        self.x_lengths = []
        self.word2idx_mapping = word2index

        # Define word2idx and idx2word as empty dictionaries
        if self.word2idx_mapping:
            self.word2index = self.word2idx_mapping
        else:
            self.word2index = defaultdict(dict)
            self.index2word = defaultdict(dict)

        # Instantiate special tokens
        self.sos_token = sos_token
        self.eos_token = eos_token
        self.unk_token = unk_token
        self.pad_token = pad_token

        # Instantiate min_word_count, max_vocab_size and max_seq_len
        self.min_word_count = min_word_count
        self.max_vocab_size = max_vocab_size
        self.max_seq_len = max_seq_len

        self.use_pretrained_vectors = use_pretrained_vectors

        if self.use_pretrained_vectors:
            self.glove_path = glove_path
            self.glove_name = glove_name
            self.weights_file_name = weights_file_name

        self.build_vocab()
        self.build_vocab()


    def build_vocab(self):
        """Build the vocabulary, filter dataset sequences and create t

        """
        # Create a dictionary that maps words to their count
        self.word_count = self.word2count()

        # Trim the vocabulary
        # Get rid of out-of-vocabulary words from the dataset
        if self.min_word_count or self.max_vocab_size:
            self.trimVocab()
            self.trimDatasetVocab()

        # Trim sequences in terms of length
        if self.max_seq_len:
            if self.x_lengths:
                self.trimSeqLen()

            else:
                # Calculate sequences lengths
                self.x_lengths = [len(seq.split()) for seq in self.dat

                if self.target_col:
                    self.y_lengths = [len(seq.split()) for seq in self

                self.trimSeqLen()


        # Map each tokens to index
        if not self.word2idx_mapping:
            self.mapWord2index()

        # Crate index2word mapping
        self.index2word = {index: word for word, index in self.word2in

        # Map dataset tokens to indices
        self.mapWords2indices()

        # Create weights matrix based on Glove vectors
        if self.use_pretrained_vectors:
            self.glove_vectors()


    def word2count(self):
        """Count the number of words occurrences.

        """
        # Instantiate the Counter object
        word_count = Counter()

        # Iterate through the dataset and count tokens
        for line in self.dataset[:, 0]:
            word_count.update(line.split())

            # Include strings from target column
            if self.target_col:
                for line in self.dataset[:, self.target_col]:
                    word_count.update(line.split())

        return word_count


    def trimVocab(self):
        """Trim the vocabulary in terms of the minimum word count or t

        """
        # Trim the vocabulary in terms of the minimum word count
        if self.min_word_count and not self.max_vocab_size:
            # If min_word_count <= 1, use the quantile approach
            if self.min_word_count <= 1:
                # Create the list of words count
                word_stat = [count for count in self.word_count.values
                # Calculate the quantile of words count
                quantile = int(np.quantile(word_stat, self.min_word_co
                print('Trimmed vocabulary using as mininum count threa
                      format(self.min_word_count, quantile))
                # Filter words using quantile threshold
                self.trimmed_word_count = {word: count for word, count
            # If min_word_count > 1 use standard approach
            else:
                # Filter words using count threshold
                self.trimmed_word_count = {word: count for word, count
                                   if count >= self.min_word_count}
                print('Trimmed vocabulary using as minimum count threa

        # Trim the vocabulary in terms of its maximum size
        elif self.max_vocab_size and not self.min_word_count:
            self.trimmed_word_count = {word: count for word, count in
            print('Trimmed vocabulary using maximum size of: {}'.forma
        else:
            raise ValueError('Use min_word_count or max_vocab_size, no

        print('{}/{} tokens has been retained'.format(len(self.trimmed
                                                     len(self.word_cou


    def trimDatasetVocab(self):
        """Get rid of rare words from the dataset sequences.
        """
        for row in range(self.dataset.shape[0]):
            trimmed_x = [word for word in self.dataset[row, 0].split()
            self.x_lengths.append(len(trimmed_x))
            self.dataset[row, 0] = ' '.join(trimmed_x)
        print('Trimmed input strings vocabulary')

        if self.target_col:
            for row in range(self.dataset.shape[0]):
                trimmed_y = [word for word in self.dataset[row, self.t
                             if word in self.trimmed_word_count.keys()
                self.y_lengths.append(len(trimmed_y))
                self.dataset[row, self.target_col] = ' '.join(trimmed_
            print('Trimmed target strings vocabulary')


    def trimSeqLen(self):
        """Trim dataset sequences in terms of the length.

        """
        if self.max_seq_len <= 1:
            x_threshold = int(np.quantile(self.x_lengths, self.max_sec
            if self.target_col:
                y_threshold = int(np.quantile(self.y_lengths, self.max
        else:
            x_threshold = self.max_seq_len
            if self.target_col:
                y_threshold =  self.max_seq_len

        if self.target_col:
            for row in range(self.dataset.shape[0]):
                x_truncated = ' '.join(self.dataset[row, 0].split()[:x
                if self.x_lengths[row] > x_threshold else self.dataset

                # Add 1 if the EOS token is going to be added to the s
                self.x_lengths[row] = len(x_truncated.split()) if not 
                                      len(x_truncated.split()) + 1

                self.dataset[row, 0] = x_truncated

                y_truncated = ' '.join(self.dataset[row, self.target_c
                if self.y_lengths[row] > y_threshold else self.dataset

                # Add 1 or 2 to the length to inculde special tokens
                y_length = len(y_truncated.split())
                if self.sos_token and not self.eos_token:
                    y_length = len(y_truncated.split()) + 1
                elif self.eos_token and not self.sos_token:
                    y_length = len(y_truncated.split()) + 1
                elif self.sos_token and self.eos_token:
                    y_length = len(y_truncated.split()) + 2

                self.y_lengths[row] = y_length

                self.dataset[row, self.target_col] = y_truncated

            print('Trimmed input sequences lengths to the length of: {
            print('Trimmed target sequences lengths to the length of: 

        else:
            for row in range(self.dataset.shape[0]):

                x_truncated = ' '.join(self.dataset[row, 0].split()[:x
                if self.x_lengths[row] > x_threshold else self.dataset

                # Add 1 if the EOS token is going to be added to the s
                self.x_lengths[row] = len(x_truncated.split()) if not 
                                      len(x_truncated.split()) + 1

                self.dataset[row, 0] = x_truncated

            print('Trimmed input sequences lengths to the length of: {


    def mapWord2index(self):
        """Populate vocabulary word2index dictionary.

        """
        # Add special tokens as first elements in word2index dictionar
        token_count = 0
        for token in [self.pad_token, self.sos_token, self.eos_token, 
            if token:
                self.word2index[token] = token_count
                token_count += 1

        # If vocabulary is trimmed, use trimmed_word_count
        if self.min_word_count or self.max_vocab_size:
            for key in self.trimmed_word_count.keys():
                self.word2index[key] = token_count
                token_count += 1

        # If vocabulary is not trimmed, iterate through dataset
        else:
            for line in self.dataset.iloc[:, 0]:
                for word in line.split():
                    if word not in self.word2index.keys():
                        self.word2index[word] = token_count
                        token_count += 1
            # Include strings from target column
            if self.target_col:
                for line in self.dataset.iloc[:, self.target_col]:
```

```
                for line in self.dataset.iloc[:, self.target_col]:
                    for word in line.split():
                        if word not in self.word2index.keys():
                            self.word2index[word] = token_count
                            token_count += 1

        self.word2index.default_factory = lambda: self.word2index[self


    def mapWords2indices(self):
        """Iterate through the dataset to map each word to its corresp
        Use special tokens if specified.

        """
        for row in range(self.dataset.shape[0]):
            words2indices = []
            for word in self.dataset[row, 0].split():
                words2indices.append(self.word2index[word])

            # Append the end of the sentence token
            if self.eos_token:
                words2indices.append(self.word2index[self.eos_token])

            self.dataset[row, 0] = np.array(words2indices)

        # Map strings from target column
        if self.target_col:
            for row in range(self.dataset.shape[0]):
                words2indices = []

                # Insert the start of the sentence token
                if self.sos_token:
                    words2indices.append(self.word2index[self.sos_toke

                for word in self.dataset[row, self.target_col].split()
                    words2indices.append(self.word2index[word])


                # Append the end of the sentence token
                if self.eos_token:
                    words2indices.append(self.word2index[self.eos_toke

                self.dataset[row, self.target_col] = np.array(words2in

        print('Mapped words to indices')


    def glove_vectors(self):
        """ Read glove vectors from a file, create the matrix of weigh
        Save the weights matrix to the numpy file.
        """
        # Load Glove word vectors to the pandas dataframe
        try:
            gloves = pd.read_csv(self.glove_path + self.glove_name, se
        except FileNotFoundError:
            print('File: {} not found in: {} directory'.format(self.gl

        # Map Glove words to vectors
        print('Start creating glove_word2vector dictionary')
        self.glove_word2vector = gloves.T.to_dict(orient='list')

        # Extract embedding dimension
        emb_dim = int(re.findall('\d+' ,self.glove_name)[-1])
        # Length of the vocabulary
        matrix_len = len(self.word2index)
        # Initialize the weights matrix
        weights_matrix = np.zeros((matrix_len, emb_dim))
        words_found = 0

        # Populate the weights matrix
        for word, index in self.word2index.items():
            try:
                weights_matrix[index] = np.array(self.glove_word2vecto
                words_found += 1
            except KeyError:
                # If vector wasn't found in Glove, initialize random v
                weights_matrix[index] = np.random.normal(scale=0.6, si

        # Save the weights matrix into numpy file
        np.save(self.weights_file_name, weights_matrix, allow_pickle=F

        # Delete glove_word2vector variable to free the memory
        del self.glove_word2vector

        print('Extracted {}/{} of pre-trained word vectors.'.format(wo
        print('{} vectors initialized to random numbers'.format(matrix
        print('Weights vectors saved into {}'.format(self.weights_file
```

Now that the Vocab class is ready, to test its functionality, firstly we have to load the dataset that will be processed and used to build the vocabulary.

In [21]:
```
# Load the training set
train_dataset = pd.read_csv('drugreview/drugreview_feat_clean/train_fe
                      usecols=['clean_review', 'subjectivity', 'polari
                      dtype={'clean_review': str, 'label': np.int16})
```

In [22]:

```
# Change the columns order
train_dataset = train_dataset[['clean_review', 'subjectivity', 'polari
```

In [23]:

```
# Display the first 5 rows from the dataset
train_dataset = train_dataset.dropna()
train_dataset.head()
```

Out[23]:

	clean_review	subjectivity	polarity	word_count	rating
1	okay anxiety gotten worse past couple years po...	0.4067	0.12980	150.0	9
6	reading possible effects scary medicine gave l...	0.5347	0.07983	90.0	10
9	clonazepam effective controlling agitation pro...	0.6855	0.23700	118.0	10
11	experienced effects considering anorexia nervo...	0.5750	0.50630	47.0	6
12	i&#039;ve gianvi months skin clear didn&#039;t...	0.3894	-0.10710	54.0	10

Below we will instantiate the Vocab class, that will cause that the dataset processing begins. After it finished we will be able to access vocab attributes to check out whether all objects are created properly.

In [24]:

```
train_vocab = Vocab(train_dataset, target_col=None, word2index=None, s
                    unk_token='<UNK>', pad_token='<PAD>', min_word_cou
                    use_pretrained_vectors=True, glove_path='glove/',
                    weights_file_name='glove/weights_train.npy')
```

```
Trimmed vocabulary using maximum size of: 20000
20000/39267 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 54
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 15330/20004 of pre-trained word vectors.
4674 vectors initialized to random numbers
Weights vectors saved into glove/weights_train.npy
```

In [25]:

```
# Depict the first dataset sequence
train_vocab.dataset[0][0]
```

Out[25]:

```
array([  559,    32,   292,    91,   148,   137,     7,   216,  5961,
        3718,   859,   649,  2255,  3327, 12461,  2200,   328,     9,
         171,  2269,   479,  1814,    27,   286,  5271,   351,    61,
         469,    20,   516, 19345,  5961,   448,   859,   177,    76,
       14777,    14,  1910,  5961,  2086,   139,  1062,    14,   290,
         339,   149,  9921,   143, 14778,   499,  2959,   409,  2095,
           2])
```

In [27]:

```
# Load the validation set
val_dataset = pd.read_csv('drugreview/drugreview_feat_clean/val_feat_c
                          usecols=['clean_review', 'subjectivity', 'polari
                          dtype={'clean_review': str, 'label': np.int16})
```

In [28]:

```
# Change the columns order
val_dataset = val_dataset[['clean_review', 'subjectivity', 'polarity',
```

In [29]:

```
# Display the first 5 rows from the dataset
val_dataset = val_dataset.dropna()
val_dataset.head()
```

Out[29]:

	clean_review	subjectivity	polarity	word_count	rating
1	4yrs having nexaplon implant mental physical h...	0.4553	0.1217	137.0	1
4	l5 s1 lumbar herniated disc surgery weeks surg...	0.3792	0.1459	69.0	10
5	far lot acne clear tea tree broke decided birt...	0.5540	0.2375	85.0	5
6	insulin works fine trouble pen pain pen jammed...	0.5500	-0.0958	47.0	4
7	nexplanon option work iud painful insert pills...	0.4426	-0.0353	135.0	7

In [30]:
```
val_vocab = Vocab(val_dataset, target_col=None, word2index=train_vocab
                  unk_token='<UNK>', pad_token='<PAD>', min_word_count
                  use_pretrained_vectors=True, glove_path='glove/', gl
                  weights_file_name='glove/weights_val.npy')
```

```
Trimmed vocabulary using maximum size of: 20000
19770/19770 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 55
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 17832/26512 of pre-trained word vectors.
8680 vectors initialized to random numbers
Weights vectors saved into glove/weights_val.npy
```

In [31]:
```
# Depict the first dataset sequence
val_vocab.dataset[10][0]
```

Out[31]:
```
array([1178, 1344,  845,   16,  814,   19,   17,   14,   16, 1230,   4
43,
        229,  109,    9,  257,  554,  235,    3,  317,  482, 1016,   3
19,
        607,   36,  607,   27,  607,  249,  548,   16,  469,   75,   3
60,
        512,  299,  293, 1028,  183,   17,   34,    2])
```

The next task to do is to create the BatchIterator class that will enable to sort dataset examples, generate batches of input and output variables, apply padding if required and be capable of iterating through all created batches. To warrant that the padding operation within one batch is limited, we have to sort examples within entire dataset according to sequences lengths, so that each batch will contain sequences with the most similar lengths and the number of padding tokens will be reduced.

In [32]:
```
class BatchIterator:

    """The BatchIterator class is responsible for:
    Sorting dataset examples.
    Generating batches.
    Sequence padding.
    Enabling BatchIterator instance to iterate through all batches.

    Parameters
    ----------
    dataset : pandas.DataFrame or numpy.ndarray
        If vocab_created is False, pass Pandas or numpy dataset contai
        to process and target non-string variable as last column. Othe
    batch_size: int, optional (default=None)
```

```
batch_size: int, optional (default=None)
    The size of the batch. By default use batch_size equal to the
vocab_created: boolean, optional (default=True)
    Whether the vocab object is already created.
vocab: Vocab object, optional (default=None)
    Use if vocab_created = True, pass the vocab object.
target_col: int, optional (default=None)
    Column index refering to targets strings to process.
word2index: dict, optional (default=None)
    Specify the word2index mapping.
sos_token: str, optional (default='<SOS>')
    Use if vocab_created = False. Start of sentence token.
eos_token: str, optional (default='<EOS>')
    Use if vocab_created = False. End of sentence token.
unk_token: str, optional (default='<UNK>')
    Use if vocab_created = False. Token that represents unknown wo
pad_token: str, optional (default='<PAD>')
    Use if vocab_created = False. Token that represents padding.
min_word_count: float, optional (default=5)
    Use if vocab_created = False. Specify the minimum word count t
    if value > 1 was passed. If min_word_count <= 1 then keep all
    quantile=min_word_count of the count distribution.
max_vocab_size: int, optional (default=None)
    Use if vocab_created = False. Maximum size of the vocabulary.
max_seq_len: float, optional (default=0.8)
    Use if vocab_created = False. Specify the maximum length of th
    max_seq_len > 1. If max_seq_len <= 1 then set the maximum leng
    quantile=max_seq_len of lengths distribution. Trimm all sequer
    than max_seq_len.
use_pretrained_vectors: boolean, optional (default=False)
    Use if vocab_created = False. Whether to use pre-trained Glove
glove_path: str, optional (default='Glove/')
    Use if vocab_created = False. Path to the directory that conta
glove_name: str, optional (default='glove.6B.100d.txt')
    Use if vocab_created = False. Name of the Glove word vectors f
    glove.6B.50d.txt
    glove.6B.100d.txt
    glove.6B.200d.txt
    glove.6B.300d.txt
    glove.twitter.27B.50d.txt
    To use different word vectors, load their file to the vectors
weights_file_name: str, optional (default='Glove/weights.npy')
    Use if vocab_created = False. The path and the name of the num

Raises
-------
ValueError('Use min_word_count or max_vocab_size, not both!')
    If both: min_word_count and max_vocab_size are provided.
FileNotFoundError
    If the glove file doesn't exist in the given directory.
TypeError('Cannot convert to Tensor. Data type not recognized')
```

```
        If the data type of the sequence cannot be converted to the Te

    Yields
    ------
    dict
        Dictionary that contains variables batches.

    """


    def __init__(self, dataset, batch_size=None, vocab_created=False,
                 sos_token='<SOS>', eos_token='<EOS>', unk_token='<UNK>',
                 max_vocab_size=None, max_seq_len=0.8, use_pretrained_vect
                 glove_name='glove.6B.100d.txt', weights_file_name='glove/

        # Create vocabulary object
        if not vocab_created:
            self.vocab = Vocab(dataset, target_col=target_col, word2in
                               unk_token=unk_token, pad_token=pad_toke
                               max_vocab_size=max_vocab_size, max_seq_
                               use_pretrained_vectors=use_pretrained_v
                               glove_name=glove_name, weights_file_nam

            # Use created vocab.dataset object
            self.dataset = self.vocab.dataset

        else:
            # If vocab was created then dataset should be the vocab.da
            self.dataset = dataset
            self.vocab = vocab

        self.target_col = target_col

        self.word2index = self.vocab.word2index

        # Define the batch_size
        if batch_size:
            self.batch_size = batch_size
        else:
            # Use the length of dataset as batch_size
            self.batch_size = len(self.dataset)

        self.x_lengths = np.array(self.vocab.x_lengths)

        if self.target_col:
            self.y_lengths = np.array(self.vocab.y_lengths)

        self.pad_token = self.vocab.word2index[pad_token]

        self.sort_and_batch()
    def sort_and_batch(self):
        """ Sort examples within entire dataset, then perform batching

        """
        # Extract row indices sorted according to lengths
        if not self.target_col:
            sorted_indices = np.argsort(self.x_lengths)
        else:
            sorted_indices = np.lexsort((self.y_lengths, self.x_length

        # Sort all sets
        self.sorted_dataset = self.dataset[sorted_indices[::-1]]
        self.sorted_x_lengths = np.flip(self.x_lengths[sorted_indices]

        if self.target_col:
            self.sorted_target = self.sorted_dataset[:, self.target_co
            self.sorted_y_lengths = np.flip(self.x_lengths[sorted_indi
        else:
            self.sorted_target = self.sorted_dataset[:, -1]

        # Initialize input, target and lengths batches
        self.input_batches = [[] for _ in range(self.sorted_dataset.sh

        self.target_batches, self.x_len_batches = [], []

        self.y_len_batches = [] if self.target_col else None

        # Create batches
        for i in range(self.sorted_dataset.shape[1]-1):
            # The first column contains always sequences that should b
            if i == 0:
                self.create_batches(self.sorted_dataset[:, i], self.ir
            else:
                self.create_batches(self.sorted_dataset[:, i], self.ir

        if self.target_col:
            self.create_batches(self.sorted_target, self.target_batche
            self.create_batches(self.sorted_y_lengths, self.y_len_batc
        else:
            self.create_batches(self.sorted_target, self.target_batche

        self.create_batches(self.sorted_x_lengths, self.x_len_batches)

        # Shuffle batches
        self.indices = np.arange(len(self.input_batches[0]))
        np.random.shuffle(self.indices)

        for j in range(self.sorted_dataset.shape[1]-1):
            self.input_batches[j] = [self.input_batches[j][i] for i in
        self.target_batches = [self.target_batches[i] for i in self.ir
        self.x_len_batches = [self.x_len_batches[i] for i in self.indi

        if self.target_col:
            self.y_len_batches = [self.y_len_batches[i] for i in self.

        print('Batches created')


    def create_batches(self, sorted_dataset, batches, pad_token=-1):
        """ Convert each sequence to pytorch Tensor, create batches ar

        """
        # Calculate the number of batches
        n_batches = int(len(sorted_dataset)/self.batch_size)

        # Create list of batches
        list_of_batches = np.array([sorted_dataset[i*self.batch_size:(
                                    for i in range(n_batches+1)])

        # Convert each sequence to pytorch Tensor
        for batch in list_of_batches:
            tensor_batch = []
            tensor_type = None
            for seq in batch:
                # Check seq data type and convert to Tensor
                if isinstance(seq, np.ndarray):
                    tensor = torch.LongTensor(seq)
                    tensor_type = 'int'
                elif isinstance(seq, np.integer):
                    tensor = torch.LongTensor([seq])
                    tensor_type = 'int'
                elif isinstance(seq, np.float):
                    tensor = torch.FloatTensor([seq])
                    tensor_type = 'float'
                elif isinstance(seq, int):
                    tensor = torch.LongTensor([seq])
                    tensor_type = 'int'
                elif isinstance(seq, float):
                    tensor = torch.FloatTensor([seq])
                    tensor_type = 'float'
                else:
                    raise TypeError('Cannot convert to Tensor. Data ty

                tensor_batch.append(tensor)
            if pad_token != -1:
                # Pad required sequences
                pad_batch = torch.nn.utils.rnn.pad_sequence(tensor_bat
                batches.append(pad_batch)
            else:
                if tensor_type == 'int':
                    batches.append(torch.LongTensor(tensor_batch))
                else:
                    batches.append(torch.FloatTensor(tensor_batch))


    def __iter__(self):
        """ Iterate through batches.

        """
        # Create a dictionary that holds variables batches to yield
        to_yield = {}

        # Iterate through batches
        for i in range(len(self.input_batches[0])):
            feat_list = []
            for j in range(1, len(self.input_batches)):
                feat = self.input_batches[j][i].type(torch.FloatTensor
                feat_list.append(feat)

            if feat_list:
                input_feat = torch.cat(feat_list, dim=1)
                to_yield['input_feat'] = input_feat

            to_yield['input_seq'] = self.input_batches[0][i]

            to_yield['target'] = self.target_batches[i]
            to_yield['x_lengths'] = self.x_len_batches[i]

            if self.target_col:
                to_yield['y_length'] = self.y_len_batches[i]


            yield to_yield


    def __len__(self):
        """ Return iterator length.

        """
        return len(self.input_batches[0])
```

Now we are going to instantiate the BatchIterator class and check out whether all tasks were conducted correctly.

In [33]:
```
train_iterator = BatchIterator(train_dataset, batch_size=32, vocab_cre
                               word2index=None, sos_token='<SOS>', eos
                               pad_token='<PAD>', min_word_count=5, ma
                               use_pretrained_vectors=True, glove_path
                               weights_file_name='glove/weights_train.
```

```
Trimmed vocabulary using as minimum count threashold: count = 5.00
10974/39267 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 53
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 9623/10978 of pre-trained word vectors.
1355 vectors initialized to random numbers
Weights vectors saved into glove/weights_train.npy
Batches created
```

In [34]:
```
# Print the size of first input batch
len(train_iterator.input_batches[0][0])
```

Out[34]:
```
32
```

In [35]:
```
# Run the BatchIterator and print the first set of batches
for batches in train_iterator:
    pprint(batches)
    break
```

```
{'input_feat': tensor([[ 4.1670e-01,  1.3540e-02,  1.1400e+02],
        [ 3.3620e-01,  8.2600e-02,  1.3000e+02],
        [ 5.0440e-01, -1.2820e-01,  1.0700e+02],
        [ 5.5660e-01,  2.7470e-01,  1.4200e+02],
        [ 4.6830e-01,  1.0500e-01,  1.4000e+02],
        [ 4.2400e-01,  4.6300e-02,  1.1600e+02],
        [ 1.9340e-01,  4.3330e-02,  1.0500e+02],
        [ 4.5000e-01, -9.3800e-03,  9.6000e+01],
        [ 5.6250e-01,  2.6380e-02,  1.3000e+02],
        [ 5.9600e-01,  1.2054e-01,  1.3900e+02],
        [ 6.0200e-01, -2.3820e-01,  9.9000e+01],
        [ 4.5480e-01,  7.3360e-02,  1.0300e+02],
        [ 5.3900e-01, -7.2140e-02,  1.2600e+02],
        [ 5.4600e-01,  1.1456e-01,  1.1600e+02],
        [ 5.7600e-01, -1.3200e-01,  1.4100e+02],
        [ 7.0360e-01, -2.2060e-01,  1.3100e+02],
        [ 6.1230e-01,  1.8890e-02,  1.0800e+02],
        [ 4.3950e-01,  1.6460e-01,  1.2000e+02],
        [ 5.1370e-01, -2.7500e-01,  1.1200e+02],
        [ 2.9440e-01,  8.3300e-02,  1.2200e+02],
        [ 5.2050e-01,  5.0630e-02,  9.3000e+01],
        [ 6.2740e-01, -9.9370e-02,  8.6000e+01],
        [ 4.6580e-01,  1.9710e-01,  1.3500e+02],
        [ 3.7820e-01,  1.7760e-01,  1.0900e+02],
        [ 5.1800e-01,  2.4130e-01,  1.0600e+02],
        [ 4.5000e-01,  5.0000e-02,  1.0100e+02],
        [ 5.5800e-01,  1.9170e-01,  1.0700e+02],
        [ 5.0100e-01, -1.2980e-01,  1.1500e+02],
        [ 4.1280e-01, -6.8170e-03,  1.0100e+02],
        [ 5.2500e-01,  5.9720e-02,  1.1000e+02],
        [ 5.6900e-01,  1.6430e-01,  1.0200e+02],
        [ 4.2400e-01, -1.0376e-02,  1.1800e+02]]),
 'input_seq': tensor([[ 765, 7714,  316,  ...,   95, 1661,    2],
        [ 109,   10, 2294,  ...,   67, 1106,    2],
        [ 278, 5644,  141,  ..., 7345, 2732,    2],
        ...,
        [  76, 2843,  472,  ..., 1171,   83,    2],
        [ 202, 3966,   31,  ..., 1252, 4497,    2],
        [ 168,  217,  954,  ...,  264,   76,    2]]),
 'target': tensor([ 8, 10,  1,  8,  7,  9,  7,  5,  7,  9,  9,  1,  9
, 10,  9, 10, 10, 10,
         4, 10,  3,  8,  8, 10,  6,  1, 10,  9,  8,  9, 10,  1]),
 'x_lengths': tensor([47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47,
47, 47, 47, 47, 47, 47,
        47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47])}
```

In [17]:

```
val_iterator = BatchIterator(val_dataset, batch_size=32, vocab_createc
                             word2index=train_iterator.word2index, sos
                             unk_token='<UNK>', pad_token='<PAD>', mir
                             max_seq_len=0.8, use_pretrained_vectors=T
                             glove_name='glove.6B.100d.txt', weights_f
```

```
Trimmed vocabulary using as minimum count threashold: count = 5.00
3292/11853 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 50
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 5884/6402 of pre-trained word vectors.
518 vectors initialized to random numbers
Weights vectors saved into glove/weights_val.npy
Batches created
```

In [18]:

```
# Run the BatchIterator and print the first set of batches
for batches in val_iterator:
    pprint(batches)
    break
```

```
{'input_feat': tensor([[ 3.5550e-01, -3.1250e-02,  9.2000e+01],
        [ 5.9860e-01, -1.5274e-02,  9.0000e+01],
        [ 6.1230e-01,  2.7900e-01,  1.0800e+02],
        [ 3.5840e-01,  2.0000e-01,  8.5000e+01],
        [ 5.0700e-01,  4.6960e-03,  8.4000e+01],
        [ 6.3000e-01,  1.2000e-01,  9.0000e+01],
        [ 4.7270e-01, -1.6370e-01,  9.5000e+01],
        [ 3.4550e-01,  9.8750e-02,  1.0100e+02],
        [ 5.9700e-01,  6.3900e-02,  1.0100e+02],
        [ 2.8500e-01, -3.1680e-02,  8.9000e+01],
        [ 6.3230e-01,  1.4360e-01,  8.9000e+01],
        [ 5.9570e-01, -2.0560e-01,  9.1000e+01],
        [ 2.0080e-01,  1.3750e-02,  1.2000e+02],
        [ 6.4550e-01,  4.4100e-02,  1.3000e+02],
        [ 3.3470e-01,  2.3600e-02,  8.0000e+01],
        [ 4.4380e-01, -1.0754e-01,  1.0500e+02],
        [ 6.6650e-01, -8.3300e-02,  5.6000e+01],
        [ 6.5900e-01, -2.5000e-02,  8.1000e+01],
        [ 6.4160e-01,  1.5760e-01,  8.2000e+01],
        [ 5.7800e-01, -1.8170e-02,  7.6000e+01],
        [ 5.3300e-01,  8.1670e-02,  7.7000e+01],
        [ 7.2700e-01,  7.7100e-02,  7.8000e+01],
        [ 4.7970e-01, -1.2560e-01,  7.5000e+01],
        [ 4.0040e-01,  2.3520e-01,  9.8000e+01],
        [ 5.5400e-01,  1.8750e-01,  9.1000e+01],
        [ 7.1800e-01,  1.8130e-01,  9.2000e+01],
        [ 4.1380e-01, -3.1770e-02,  1.1600e+02],
        [ 6.7770e-01, -3.8880e-02,  1.1500e+02],
        [ 6.4300e-01,  7.1400e-03,  9.5000e+01],
        [ 4.6000e-01,  1.3000e-01,  7.8000e+01],
        [ 6.3230e-01,  3.0400e-01,  1.0400e+02],
        [ 3.3400e-01,  1.0236e-01,  1.1700e+02]]),
 'input_seq': tensor([[ 568,  914,  503,  ..., 4028,  422,    2],
        [ 138,   65,  820,  ..., 1253,  274,    2],
        [4774,  136,   36,  ..., 1396,  544,    2],
        ...,
        [2822,  798,  194,  ..., 1132,  139,    2],
        [5005, 2529,  335,  ...,  227, 2682,    2],
        [2035,  719,  274,  ..., 1231,  213,    2]]),
 'target': tensor([ 1,  9,  9,  8,  8, 10,  1,  9,  4,  8,  9,  1,  8
,  1,  1,  4,  8, 10,
         8,  7, 10,  8,  1, 10,  5,  9, 10,  5, 10,  6, 10, 10]),
 'x_lengths': tensor([36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36,
36, 36, 36, 36, 36, 36,
        36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36])}
```