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 [1]:
```
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/
Fasttext word vectors can be found under the link: https://fasttext.cc/docs/en/english-vectors.html

In [2]:
```
class Vocab:

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

```
    Filtering dataset in terms of the rare words occurrence and sentences len
    Mapping words to their numerical representation (word2index) and reverse
    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 strings
        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 vocabul
        If min_word_count <= 1 then keep all words whose count is greater tha
        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 max_seq
        the maximum length to value corresponding to quantile=max_seq_len of
        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 vectors
    glove_name: str, optional (default='glove.6B.100d.txt')
        Name of the Glove word vectors file. Available pretrained vectors:
        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 directo
    weights_file_name: str, optional (default='Glove/weights.npy')
        The path and the name of the numpy file to which save weights vectors

    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_token='
             pad_token='<PAD>', min_word_count=5, max_vocab_size=None, max_se
             use_pretrained_vectors=False, glove_path='glove/', glove_name='g
             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()


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

```
        """
        # 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.dataset[:,

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

                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.word2index.ite

        # 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 the voca

    """
    # 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_count))
            print('Trimmed vocabulary using as mininum count threashold:
                  format(self.min_word_count, quantile))
            # Filter words using quantile threshold
            self.trimmed_word_count = {word: count for word, count in sel
        # If min_word_count > 1 use standard approach
        else:
            # Filter words using count threshold
            self.trimmed_word_count = {word: count for word, count in sel
                                       if count >= self.min_word_count}
            print('Trimmed vocabulary using as minimum count threashold:

    # 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 self.wo
        print('Trimmed vocabulary using maximum size of: {}'.format(self.
    else:
        raise ValueError('Use min_word_count or max_vocab_size, not both!

    print('{}/{} tokens has been retained'.format(len(self.trimmed_word_c
                                                  len(self.word_count.keys


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() if wor
        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.target_c
```

```
                            if word in self.trimmed_word_count.keys()]
                self.y_lengths.append(len(trimmed_y))
                self.dataset[row, self.target_col] = ' '.join(trimmed_y)
            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_seq_len))
            if self.target_col:
                y_threshold = int(np.quantile(self.y_lengths, self.max_seq_le
        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_thresh
                if self.x_lengths[row] > x_threshold else self.dataset[row, 0

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

                self.dataset[row, 0] = x_truncated

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

                # 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: {}'.form
            print('Trimmed target sequences lengths to the length of: {}'.for

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

                x_truncated = ' '.join(self.dataset[row, 0].split()[:x_thresh
```

```
                if self.x_lengths[row] > x_threshold else self.dataset[row, 0

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

                self.dataset[row, 0] = x_truncated

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


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

        """
        # Add special tokens as first elements in word2index dictionary
        token_count = 0
        for token in [self.pad_token, self.sos_token, self.eos_token, self.un
            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 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.unk_to


    def mapWords2indices(self):
        """Iterate through the dataset to map each word to its corresponding
        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_token])

                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_token])

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

        print('Mapped words to indices')


    def glove_vectors(self):
        """ Read glove vectors from a file, create the matrix of weights mapp
        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, sep=" ",
        except FileNotFoundError:
            print('File: {} not found in: {} directory'.format(self.glove_nam

        # 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_word2vector[word]
                words_found += 1
            except KeyError:
                # If vector wasn't found in Glove, initialize random vector
                weights_matrix[index] = np.random.normal(scale=0.6, size=(emb

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

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

        print('Extracted {}/{} of pre-trained word vectors.'.format(words_fou
        print('{} vectors initialized to random numbers'.format(matrix_len -
        print('Weights vectors saved into {}'.format(self.weights_file_name))
```

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 [3]:
```
# Load the training set
train_dataset = pd.read_csv('dataset/drugreview_feat_clean/train_feat_clean.c
                            usecols=['clean_review', 'subjectivity', 'polarity', 'w
                            dtype={'clean_review': str, 'label': np.int16})
```

In [4]:
```
# Change the columns order
train_dataset = train_dataset[['clean_review', 'subjectivity', 'polarity', 'w
```

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

Out[5]:

	clean_review	subjectivity	polarity	word_count	rating
2	young suffering severe extreme neck pain resul...	0.4026	-0.04166	120.0	10
5	found work helping good nights sleep don'...	0.6000	0.70000	22.0	7
9	given medication gastroenterologist office wor...	0.0000	0.00000	36.0	1
12	recently laparoscopic hysterectomy know anesth...	0.6970	-0.29400	98.0	10
13	mirena year experienced effects effects watch ...	0.9000	0.80000	37.0	1

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 [6]:
```
train_vocab = Vocab(train_dataset, target_col=None, word2index=None, sos_toke
                    unk_token='<UNK>', pad_token='<PAD>', min_word_count=None
                    use_pretrained_vectors=True, glove_path='glove/', glove_n
                    weights_file_name='glove/weights.npy')
```

```
Trimmed vocabulary using maximum size of: 5000
5000/21861 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 51
Mapped words to indices
Start creating glove_word2vector dictionary
Extracted 4684/5004 of pre-trained word vectors.
320 vectors initialized to random numbers
Weights vectors saved into glove/weights.npy
```

In [7]:
```
# Depict the first dataset sequence
train_vocab.dataset[0][0]
```

Out[7]:
```
array([ 964,  395,   60,  269,  365,    9, 3183,  965,  107,   81,  304,
       1145,  771, 1014,  638, 4223,  759, 2614, 4494,  771,  771,  257,
        771,  240,   28,  339,    9, 2164,  247,   12,  302, 1209, 1770,
       1349,  411,   14,   71,   36,  848,  124,  628,  638, 2041,  609,
       2115,  966,  175, 1853, 1570, 1570, 1079,    2])
```

In [8]:
```
# Load the validation set
val_dataset = pd.read_csv('dataset/drugreview_feat_clean/val_feat_clean.csv',
                          usecols=['clean_review', 'subjectivity', 'polarity', 'w
                          dtype={'clean_review': str, 'label': np.int16})
```

In [9]:
```
# Change the columns order
val_dataset = val_dataset[['clean_review', 'subjectivity', 'polarity', 'word_
```

In [10]:
```
# Display the first 5 rows from the dataset
val_dataset = val_dataset.dropna()
val_dataset.head()
```

Out[10]:

	clean_review	subjectivity	polarity	word_count	rating
0	year old son took night went deep sea fishing ...	0.1813	0.0250	66.0	7
1	daughter epiduo grade junior year work wonders...	0.4402	0.1320	128.0	9
2	i've implant months day got totally felt ...	0.5520	0.1597	148.0	8
3	wanted wait days post couldn't results am...	0.5977	0.2349	102.0	10
4	colonoscopy best prep far morning took prep pm...	0.4224	0.0782	136.0	9

In [11]:
```
val_vocab = Vocab(val_dataset, target_col=None, word2index=train_vocab.word2i
                  unk_token='<UNK>', pad_token='<PAD>', min_word_count=None, 
                  use_pretrained_vectors=False, glove_path='Glove/', glove_na
                  weights_file_name='Glove/weights.npy')
```

```
Trimmed vocabulary using maximum size of: 5000
5000/11853 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 52
Mapped words to indices
```

In [12]:
```
# Depict the first dataset sequence
val_vocab.dataset[10][0]
```

Out[12]:
```
array([709, 110, 209,   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 [13]:
```
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 containing in
    to process and target non-string variable as last column. Otherwise p
batch_size: int, optional (default=None)
    The size of the batch. By default use batch_size equal to the dataset
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 words.
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 threshol
    if value > 1 was passed. If min_word_count <= 1 then keep all words w
    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 the seque
    max_seq_len > 1. If max_seq_len <= 1 then set the maximum length to v
    quantile=max_seq_len of lengths distribution. Trimm all sequences who
    than max_seq_len.
use_pretrained_vectors: boolean, optional (default=False)
    Use if vocab_created = False. Whether to use pre-trained Glove vector
glove_path: str, optional (default='Glove/')
    Use if vocab_created = False. Path to the directory that contains fil
glove_name: str, optional (default='glove.6B.100d.txt')
    Use if vocab_created = False. Name of the Glove word vectors file. Av
    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 directo
weights_file_name: str, optional (default='Glove/weights.npy')
    Use if vocab_created = False. The path and the name of the numpy file
```

```
    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 Tensor.

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

    """


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

        # Create vocabulary object
        if not vocab_created:
            self.vocab = Vocab(dataset, target_col=target_col, word2index=wor
                               unk_token=unk_token, pad_token=pad_token, min_
                               max_vocab_size=max_vocab_size, max_seq_len=max
                               use_pretrained_vectors=use_pretrained_vectors,
                               glove_name=glove_name, weights_file_name=weigh

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

        else:
            # If vocab was created then dataset should be the vocab.dataset o
            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 and sh

        """
        # 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_lengths))

        # 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_col]
            self.sorted_y_lengths = np.flip(self.x_lengths[sorted_indices])
        else:
            self.sorted_target = self.sorted_dataset[:, -1]

        # Initialize input, target and lengths batches
        self.input_batches = [[] for _ in range(self.sorted_dataset.shape[1]-

        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 be padde
            if i == 0:
                self.create_batches(self.sorted_dataset[:, i], self.input_bat
            else:
                self.create_batches(self.sorted_dataset[:, i], self.input_bat

        if self.target_col:
            self.create_batches(self.sorted_target, self.target_batches, pad_
            self.create_batches(self.sorted_y_lengths, self.y_len_batches)
        else:
            self.create_batches(self.sorted_target, self.target_batches)

        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.i

        self.target_batches = [self.target_batches[i] for i in self.indices]
        self.x_len_batches = [self.x_len_batches[i] for i in self.indices]

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

        print('Batches created')


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

        """
        # 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:(i+1)*se
                                    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 type not

                tensor_batch.append(tensor)
            if pad_token != -1:
                # Pad required sequences
                pad_batch = torch.nn.utils.rnn.pad_sequence(tensor_batch, 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).unsqu
                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 [14]:

```
train_iterator = BatchIterator(train_dataset, batch_size=32, vocab_created=Fa
                               word2index=None, sos_token='<SOS>', eos_token=
                               pad_token='<PAD>', min_word_count=5, max_vocab
                               use_pretrained_vectors=False, glove_path='glov
                               weights_file_name='glove/weights.npy')
```

```
Trimmed vocabulary using as minimum count threashold: count = 5.00
6362/21861 tokens has been retained
Trimmed input strings vocabulary
Trimmed input sequences lengths to the length of: 52
Mapped words to indices
Batches created
```

In [15]:
```
# Print the size of first input batch
len(train_iterator.input_batches[0][0])
```

Out[15]: 32

In [16]:
```
# Run the BatchIterator and print the first set of batches
for batches in train_iterator:
    pprint(batches)
    break
```

```
{'input_feat': tensor([[ 5.6450e-01, -7.8550e-02,  6.0000e+01],
        [ 2.1880e-01, -5.6240e-02,  6.8000e+01],
        [ 5.5000e-01,  1.9370e-01,  9.0000e+01],
        [ 4.2700e-01, -2.5000e-02,  6.4000e+01],
        [ 7.7640e-01,  1.2220e-01,  8.0000e+01],
        [ 4.0500e-01,  3.5000e-02,  5.7000e+01],
        [ 6.5000e-01,  2.1670e-01,  6.7000e+01],
        [ 6.8300e-01,  3.7500e-02,  6.7000e+01],
        [ 6.3000e-01,  2.2740e-01,  7.8000e+01],
        [ 5.5220e-01, -4.1080e-02,  6.7000e+01],
        [ 3.3330e-01,  0.0000e+00,  6.5000e+01],
        [ 5.2340e-01, -5.7000e-02,  5.5000e+01],
        [ 6.8900e-01,  3.4160e-01,  6.0000e+01],
        [ 3.0000e-01, -1.9370e-01,  6.4000e+01],
        [ 4.6660e-01, -7.5000e-02,  6.8000e+01],
        [ 7.6660e-01,  2.6660e-01,  5.7000e+01],
        [ 6.1100e-01, -4.5230e-02,  5.7000e+01],
        [ 5.1270e-01,  1.8050e-02,  6.7000e+01],
        [ 8.3640e-01,  8.8000e-02,  6.0000e+01],
        [ 5.7500e-01,  2.2500e-01,  8.6000e+01],
        [ 6.0640e-01, -8.8300e-02,  5.9000e+01],
        [ 6.0640e-01, -2.6670e-02,  7.0000e+01],
        [ 4.4730e-01,  2.0280e-01,  7.6000e+01],
        [ 4.5630e-01, -1.2740e-01,  5.6000e+01],
        [ 2.0000e-01,  6.2500e-02,  5.9000e+01],
        [ 3.9580e-01,  9.0940e-02,  6.8000e+01],
        [ 6.3800e-01,  4.2580e-01,  6.7000e+01],
        [ 4.2970e-01,  5.2300e-02,  6.7000e+01],
        [ 5.5370e-01, -2.3800e-01,  7.3000e+01],
        [ 4.0000e-01,  1.3610e-01,  6.7000e+01],
        [ 2.6660e-01,  1.8330e-01,  5.8000e+01],
        [ 8.3740e-01,  6.6260e-01,  4.6000e+01]]),
 'input_seq': tensor([[  70,   65,   69,  148, 1636, 1665, 4146, 2104, 1503, 4006,  305, 1615,
           58,  201,  116,  426,   36, 2863, 1791,   61,   78, 2672, 2489,  413,
           70, 2489,    2],
```

```
        [4993,  335,   68,  891,  155, 4467, 3405, 1353, 4472,  332, 4472,  15
0,
           56,  247, 1336, 5014,  510, 3526, 2059,  132,  148,  878,   56,  89
1,
           84,  172,    2],
        [ 138, 2869,  134,   36,  257,  138,  139,  140,  702, 4894,  260, 166
5,
         2186,  274,  274,   93,  299,  335, 2718,  544,  299, 3949, 2429,  66
8,
          299, 3949,    2],
        [ 127,  388,  148,  205,  145,  181,  119,  327,  539,  245, 4280,   6
1,
          752,  253,  148,   97,  726, 1798, 1375,  826,  737,  372,   61,  40
0,
         4238,  315,    2],
        [ 448,  172,  324, 1174,  412,   30,  915, 2298,   92, 1118,  174, 438
5,
          986, 1541,  217,  204, 1541,  450,  201,  181, 4288, 3424, 3320,  33
1,
          727, 1540,    2],
        [1173, 5199,  422,  134,  787,  367,  651,  120, 4368, 1489,  458,
9,
          715, 1964,  116,   30, 1378, 1997, 3307,  642,  198, 1085,  422,  43
5,
          280,   26,    2],
        [1499,  923, 1318, 2252, 5579, 2719,   70, 2984,   70,  335,  136,   7
0,
         1499,  715, 1771,  995, 1733,  538,  464, 1597,  971,  179,   71,  22
4,
          400,  422,    2],
        [  30,   70,   80, 4791,  742,    5,  335,  718,  105,   92,  205,   3
0,
         1358, 3993,  413,  198,   84,   70,  844,  157,  130, 1954, 1259, 365
0,
         1494, 5499,    2],
        [ 377, 2622,    9,  691,  274,  202,  576, 1677,   65,  130,  670,  11
2,
          429,  337, 1116,  426,   36,    9,   70, 4069,  891, 1190,   26,  21
1,
          147,  659,    2],
        [ 205,  973,  168,  580,  112,  250, 1794, 2944, 1016,  815,  205, 144
7,
         1492, 1665,  251,   61,   78, 1374,  176, 1046,  775,  118,  262, 124
8,
           61,  268,    2],
        [  30,   70,   65,   56, 4071,  438,  437, 1053,  167,   30,  131,  83
8,
          908,  262,  252,  202,  143, 2962, 3331,  437,   65,  143, 2815, 317
6,
           65,  891,    2],
        [ 455,  110,  122,  485,  112, 2038,  677,  759,  201, 2884,  245,  20
5,
          274,  335,  321,  112,   60,  474, 1676,  650,  526,  192,  522, 384
7,
         1375,  102,    2],
        [  69,  136,  995,  862, 4168, 1733,  916, 4641, 2504,  257, 1318,   7
8,
```

```
         1737,  658,  656,  184,   36, 4642, 4643,  257,   58,  446,  148, 167
8,
          274,  223,    2],
        [  30,   70,  178,   69,  156,  127,  133,  484,   69,   70,  228,  22
9,
          178,  822, 1492, 1159,  826,   69,   64,  178,  366,  589, 2084,  14
8,
         2595,  131,    2],
        [ 205,  452, 1835,  328,   37,  217,   70,  705,  250, 1088,   69,  15
6,
          435, 1596,  204,   70,  172,  702,   36,  163, 1670,  815, 1487,  57
8,
         5949, 1493,    2],
        [3237,  172,  775, 5527,  196, 5165,  177,  143,  196, 1678, 6328,  74
3,
          196,  213, 3721, 2348, 1574,  277, 1656,   36,  206, 1535,  206, 121
1,
          928, 6196,    2],
        [ 154, 2579, 4018,  533,  210, 1044,  568,    9, 1715,  123,  576, 108
8,
           69,  422, 3162,  381, 1186,  963,  870, 1231, 1106,  875,  944, 223
4,
           70,  941,    2],
        [4461, 2595,  198, 2801, 2802, 2802,  205,  429,  337,  929,  640,  42
7,
          250,  737, 4464, 4044, 1006,  737, 3199, 2970,  143,  959,  371,  11
1,
          112, 1384,    2],
        [2586,  253, 4088,  296, 4310, 1003,  175, 2389,  253, 2087,  119, 376
2,
          718,   30, 2586,    9,   38,  166,  119, 2389,  253,  400,  705,  11
9,
         2389,  253,    2],
        [1457, 4124,  784, 4461,   30, 1503,  337,  184,  372,  327,  198, 234
8,
          448,  864,  413, 2886, 1204,   71,  483, 1091,  833,  891,  804,  80
4,
         2691,   25,    2],
        [5674,  134, 1220,  228,  155,  168,  605,  963, 1438,  118,  139,  14
0,
         4180, 5766, 1170,  130, 3376,  134,  155,  605, 3376,  130,  166,  94
4,
           13, 2717,    2],
        [ 410,  127,  831, 3173,  543,  299,  268,  357,  517,  860,  135,   3
0,
          241,  871,  184,  148,  708,  201,  745,  105,  103, 5920, 3206, 228
2,
          745,   30,    2],
        [ 213,   70,  373,  410,  831,  997, 1217,  941,  659,  204,  385,  37
3,
         1028,   38,  301,  575, 3175,  231,   17,  261,  201, 1825,  891, 141
2,
         1413, 1303,    2],
        [ 908,  740,  720,  262, 1378,  720,   30,  674, 1020,  145,  139,   7
0,
           65, 1379, 1380,   61,  143, 1381,  720,  262,  270,  666,  201,   7
0,
```

```
           58,   97,    2],
        [   5, 3180, 1720,  328, 1259, 1798, 2462, 1292,  474, 1540, 5206,  96
3,
          847, 1736,  503, 1719, 1720,  328,  421,  246, 3180, 1720,  328, 135
5,
         1614,   69,    2],
        [ 138, 2738,  134,   30,  715, 1562,   36,  105,   69,  224,  143, 274
2,
         1366,  824,  354, 3829,  555,  201,  244,  245,  119,  114,  700,  13
9,
          140,  448,    2],
        [ 378,   38, 4850, 5674,   26,   37,  952,  953,  508,  610, 1438,  13
9,
          140, 1073,  650,    9,  175,   11,    9,  133,  690,  932, 1135,  14
1,
          249,  177,    2],
        [ 202,  327,  205,   70,  202, 2631, 5803,  908, 3742, 5803,  335, 408
1,
          251,  252, 3742,  367,  280,    6, 1489,    9, 2629,   29,  400,   7
0,
         5803,  131,    2],
        [ 156,  779,  509,  538, 3013,  977,  702,  538,   36, 4897, 1308, 517
9,
         3957,   38, 4740, 1169, 2765, 4137, 1169,  386, 1957,   36, 4779, 202
3,
           36, 2618,    2],
        [ 297,   69,  156,  838,  764,  654, 5760,  578, 3976, 2453, 1966,  76
3,
         2312,  445,  945, 2171,  941,  654, 3925,    9,  175, 1179,  141,  35
6,
         2477, 3436,    2],
        [  30,  198, 3441,  429,  337,  198, 4127,  291,  646,  201,  284, 314
1,
          218,  274,    6,  521,  143, 3210,   69,  201,  385,  544,   26,  17
6,
          112, 4868,    2],
        [2006,  531, 1976, 1525,  111,  622,  136,   92, 4645,  445,  102, 378
1,
           68,  252, 1553,  130,   56,   68,  423,  203,  111, 3663, 1489,  62
2,
          547,  508,    2]]),
 'target': tensor([ 1,  1, 10,  1,  9,  1,  1,  1, 10,  3,  4,  9,  4,  3,  8,
10,  2,  9,
        10,  8,  1,  1,  1,  7,  9,  9,  9,  9,  1,  8,  6,  9]),
 'x_lengths': tensor([27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,
27, 27, 27, 27,
        27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27])}
```

In [17]:

```
val_iterator = BatchIterator(val_dataset, batch_size=32, vocab_created=False,
                             word2index=train_iterator.word2index, sos_token=
                             unk_token='<UNK>', pad_token='<PAD>', min_word_c
                             max_seq_len=0.8, use_pretrained_vectors=False, g
                             glove_name='glove.6B.100d.txt', weights_file_nam
```

```
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
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([[ 5.7230e-01, -1.3120e-01,  1.5000e+02],
        [ 6.2500e-01,  6.0550e-02,  1.3500e+02],
        [ 5.8060e-01,  1.2476e-01,  1.2100e+02],
        [ 4.6920e-01,  1.0370e-01,  1.2700e+02],
        [ 5.9900e-01, -1.0200e-02,  1.2800e+02],
        [ 5.7960e-01,  1.3400e-01,  1.2900e+02],
        [ 4.2940e-01, -1.2630e-01,  1.1600e+02],
        [ 5.6740e-01,  3.7630e-02,  1.4000e+02],
        [ 4.6240e-01, -1.5370e-01,  1.4900e+02],
        [ 4.6340e-01, -5.5540e-02,  1.1700e+02],
        [ 5.5570e-01,  1.5660e-01,  1.1800e+02],
        [ 5.2730e-01,  1.1450e-01,  1.2900e+02],
        [ 3.8770e-01,  6.1100e-02,  9.9000e+01],
        [ 5.2100e-01, -3.7880e-03,  1.3200e+02],
        [ 5.5500e-01,  1.8620e-01,  1.2400e+02],
        [ 5.6350e-01,  1.4030e-01,  1.3800e+02],
        [ 5.1700e-01, -2.0640e-01,  1.1500e+02],
        [ 5.5220e-01,  1.0090e-01,  1.4300e+02],
        [ 3.1900e-01,  2.4900e-01,  1.3800e+02],
        [ 4.4340e-01,  7.6350e-02,  1.4300e+02],
        [ 3.3700e-01,  6.5500e-02,  1.1200e+02],
        [ 6.0740e-01, -1.8690e-01,  1.3300e+02],
        [ 6.6000e-01,  1.7140e-01,  1.1400e+02],
        [ 5.1070e-01,  8.1240e-02,  1.0700e+02],
        [ 5.9200e-01, -1.1500e-01,  1.3500e+02],
        [ 4.8680e-01,  1.2500e-01,  1.3800e+02],
        [ 4.9950e-01,  2.6510e-03,  1.3700e+02],
        [ 5.1100e-01,  4.6940e-02,  1.3800e+02],
        [ 6.7600e-01,  5.3900e-01,  1.1900e+02],
        [ 3.4250e-01,  1.2890e-01,  1.3700e+02],
        [ 6.1700e-01,  1.8000e-01,  1.3500e+02],
        [ 3.0200e-01,  1.8140e-01,  1.2600e+02]]),
 'input_seq': tensor([[ 294,  238, 3491,  ..., 1954, 1698,    2],
        [ 138,  380,  325,  ...,  228,  229,    2],
        [ 138,  335, 3562,  ...,  177, 1093,    2],
        ...,
        [ 138,  136,  386,  ..., 1431,    2,    0],
        [ 684, 2385,  301,  ...,   65,    2,    0],
        [ 189, 2341,  134,  ...,  960,    2,    0]]),
 'target': tensor([ 1,  8,  7, 10,  9,  9,  3,  8,  2,  3,  3,  4, 10,  1, 10,
10,  3, 10,
         4, 10,  4, 10, 10, 10,  9,  8,  7, 10, 10,  5, 10,  9]),
 'x_lengths': tensor([49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49,
49, 49, 49, 49,
        49, 49, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48])}
```

In the next notebook we are going to create the neural network model.

In [ ]: