Boyko Borisov Migrate repository to personal git

0 contributors

1023 lines (1023 sloc) | 41.8 KB

In [1]:

```
import nltk
import string
import os
from collections import defaultdict
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import sample
from sklearn.cluster import KMeans, AgglomerativeClustering
nltk.download('stopwords')
print(torch.cuda.device_count())
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
1
```

In [2]:

```
# I used google collab for training my model, uncomment the lines below to
# connect to google drive in google colab

# from google.colab import drive
# drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [3]:

```
stop_words = set(nltk.corpus.stopwords.words("english"))
stemmer = nltk.SnowballStemmer("english", ignore_stopwords = False)
# NB corpus_root AND corpus_after_token_reversal SHOULD BE CHANGED TO MATCH
# THE CORPUS PATH ON THE SPECIFIC MACHINE
# Folder path where corpus root should be
corpus_root = r"/content/drive/MyDrive/cw2/product_reviews"
# Folder path where the reverse token corpus should be stored
corpus_after_token_reversal = r"/content/drive/MyDrive/cw2/product_reviews_proces
file_pattern = r".*"
original_corpus = nltk.corpus.PlaintextCorpusReader(corpus_root, file_pattern)
print(original_corpus.fileids())
```

```
['Canon_PowerShot_SD500.txt', 'Canon_S100.txt', 'Diaper_Champ.txt', 'Hitachi_route
'norton.txt']
```

In [4]:

```
# Core utility function for document cleaning
# Works recursively, split the text into sentences/review, then for each
# sentence/review perform cleaning
def process_doc(text, remove_punctuation, case_fold, stem,
                remove_stopwords, remove_short_tokens, tokenize_by, stem_blacklis
                remove_nonalphabetical = False):

  if (tokenize_by == "sentence"):
    sentences = nltk.RegexpTokenizer("##", gaps = True).tokenize(text)
    sentences = [process_doc(sentence, remove_punctuation, case_fold, stem,
                             remove_stopwords, remove_short_tokens, "words", stem
                 for sentence in sentences]
    return sentences
  if (tokenize_by == "sentiments"):
    sentiments = nltk.RegexpTokenizer("₩[[₩-+] [0-9] ₩]", gaps = True).tokenize(
    sentiments = [process_doc(sentiment, remove_punctuation, case_fold, stem,
                              remove_stopwords, remove_short_tokens, "words", ste
                  for sentiment in sentiments]
```

```
    return sentiments
  if (tokenize_by == "reviews"):
    reviews = nltk.RegexpTokenizer("₩[ t ₩]", gaps = True).tokenize(text)
    reviews = [process_doc(review, remove_punctuation, case_fold, stem,
                           remove_stopwords, remove_short_tokens, "words", ste
            for review in reviews]
    return reviews
  if (tokenize_by == "words"):
    words = nltk.WordPunctTokenizer().tokenize(text)
    if (remove_punctuation):
      words = [w for w in words if w not in string.punctuation and w != "..."
      # words = [w.strip("") for w in words]
    if (case_fold):
      words = [w.lower() for w in words]
    if (remove_short_tokens):
      words = [w for w in words if len(w) > 2]
    if (stem):
      words = [w if w in stem_blacklist else stemmer.stem(w) for w in words]
    if (remove_stopwords):
      words = [w for w in words if w not in stop_words and w != "n't"]
    if (remove_punctuation):
      words = [w for w in words if w not in string.punctuation and w != "..."
    if (remove_nonalphabetical):
      words = [w for w in words if w.isalpha()]
    return words

def process_corpus(corpus, remove_punctuation:bool, case_fold:bool, stem:bool,
                  remove_stopwords:bool, remove_short_tokens, tokenize_by:str, re
  docs = [word for fileid in corpus.fileids()
          for word in process_doc(corpus.raw(fileid), remove_punctuation, case
                                  stem, remove_stopwords, remove_short_tokens,
                                  tokenize_by, remove_nonalphabetical)
        ]
  return docs

def most_frequent(words, n, should_print):
  freqDist = nltk.FreqDist(words)
  most_common = freqDist.most_common(n)
  if (should_print):
    i = 1
    for (w, count) in most_common:
      print(i , w , count)
      i += 1
  return most_common

# core function for generating corpus with reversed words
# the corpus of reversed words is stored as files in the path specified by the va
# corpus_after_token_reversal
def generate_corpus_half_tokens_reversed(corpus, token_tuple_list, override_folde
  if not override_folder and os.path.exists(corpus_after_token_reversal):
    return
  if not os.path.exists(corpus_after_token_reversal):
    os.mkdir(corpus_after_token_reversal)
  # indecies_per_word = {word : list of 0s and 1s}
  # if indecies_per_word["word"][i] == 1
  #   the i-th occurrence of "word" needs to be reversed
  indecies_per_word = {}
  # pointers keeps track of how many occurrences of each word we have met
  pointers = {}
  for (word, frequency) in token_tuple_list:
    # construct an array with an equal number of 0-s and ones
    indecies = np.ones(frequency)
    indecies[:int(frequency/2)] = 0
```

```
        # shuffle it
        np.random.shuffle(indecies)
        indecies_per_word[word] = indecies
        pointers[word] = -1
    fileids = corpus.fileids()
    for fileid in fileids:
        # tokenize the document
        tokens = process_doc(corpus.raw(fileid), False, True, False, False, False
        with_reversal = []
        for token in tokens:
            if (token in indecies_per_word):
                # update the number of occurrences of the token
                pointers[token] += 1
                # determine whether to reverse the token
                if (indecies_per_word[token][pointers[token]] == 1):
                    token = token[::-1]
            with_reversal.append(token)
        doc = " ".join(with_reversal)

        f = open(os.path.join(corpus_after_token_reversal,fileid), "w")
        f.write(doc)
        f.close()
```

In [5]:

```
print("Most frequent 50 tokens in corpus after document cleaning and lemmatisatic
processed_corpus = process_corpus(original_corpus, True, True, False, True, Tr
most_frequent_tokens = most_frequent(processed_corpus, 50, True)
```

```
Most frequent 50 tokens in corpus after document cleaning and lemmatisation
1 use 353
2 phone 320
3 one 316
4 ipod 314
5 router 313
6 camera 292
7 player 269
8 get 252
9 battery 239
10 like 195
11 great 192
12 quality 176
13 good 176
14 zen 174
15 diaper 171
16 product 166
17 would 158
18 also 156
19 time 145
20 software 145
21 sound 144
22 well 138
23 really 136
24 micro 136
25 features 128
26 computer 128
27 easy 125
28 even 123
29 first 121
30 used 120
31 creative 118
32 much 115
33 better 114
34 champ 113
```

```
35 work 112
36 want 107
37 size 105
38 music 105
39 norton 104
40 little 101
41 need 100
42 pictures 99
43 works 99
44 still 97
45 buy 96
46 problem 96
47 mp3 96
48 price 91
49 life 91
50 using 91
```

In [6]:

```python
def get_all_sentences_cleaned(corpus_filepath, stemming, stopwords_removal, stem_
  corpus = nltk.corpus.PlaintextCorpusReader(corpus_filepath, file_pattern)
  out = []
  for fileid in corpus.fileids():
    sentences = process_doc(corpus.raw(fileid), True, True, stemming, stopwords_
    out.extend(sentences)
  return out


def generate_word_to_indx_and_idx_to_word(corpus):
  word_to_idx = {}
  idx_to_word = {}
  i = 0
  for sentence in corpus:
    for word in sentence:
      if (word not in word_to_idx):
        word_to_idx[word] = i
        idx_to_word[i] = word
        i += 1
  return (word_to_idx, idx_to_word)

def get_context_window_tuples(word_to_idx, sentences, window, key_words):
  tuples = []
  for sentence in sentences:
    for i in range(window, len(sentence) - window):
        context = []
        middle_word = word_to_idx[sentence[i]]
        for j in range (i - window, i + window + 1):
          if i != j:
            context.append(word_to_idx[sentence[j]])
        tuples.append((context, word_to_idx[sentence[i]]))


  return tuples


def get_skipgrams(sentences, word_to_idx, window, neg_sample_count):
  word = []
  context = []
  y = []
  for sentence in sentences:
    for i in range(len(sentence)):
      cont = [word_to_idx[sentence[idx]] for idx in range(max(0, i - window), mi
      blacklist = set(cont)
      word.extend([word_to_idx[sentence[i]]] * (len(cont)))
      context.extend(cont)
  return(word, context)
```

```
def get_batches(words, contexts, batch_size):
  shuffled_idxs = sample(range(0, len(words)), len(words))
  batches = []

  batch_word, batch_context = [], []
  for i in range(len(words)):
    idx = shuffled_idxs[i]
    batch_word.append(words[idx])
    batch_context.append(contexts[idx])
    if (i + 1) % batch_size == 0 or i + 1 == len(words):
      batches.append((
        torch.from_numpy(np.array(batch_word)),
        torch.from_numpy(np.array(batch_context))
      ))
      batch_word, batch_context = [], []
  return batches


def get_x_tensors(x_y_tuples):
  tensors = []
  for tuple in x_y_tuples:
    tensors.append(torch.tensor(tuple[0], dtype=torch.long))
  return tensors
def get_y_tensors(tuples, num_classes):
  tensors = []
  for tuple in tuples:
    tensors.append(torch.tensor([tuple[1]]))
  return tensors
```

In [7]:

```
class Word2Vec_Skipgram(nn.Module):
  def __init__(self, embedding_size, vocab_size) -> None:
        super(Word2Vec_Skipgram, self).__init__()
        self.embedding_words = nn.Embedding(vocab_size, embedding_size)
        self.linear = nn.Linear(embedding_size, vocab_size)
        self.log_softmax = nn.LogSoftmax(dim = 1)

  def forward(self, words):
      words_emb = self.embedding_words(words)
      scores = self.linear(words_emb)
      log_probs = self.log_softmax(scores)
      return log_probs
```

In [8]:

```
def train_skipgram_model(model, epochs, batch_size, learning_rate, verbose, words
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)
  loss_function = nn.NLLLoss()
  for epoch in range(epochs):
    total_loss = 0
    for inputs,targets in get_batches(words=words, contexts=contexts, batch_size
      optimizer.zero_grad()
      inputs, targets = inputs.to(device), targets.to(device)
      y_hat = model(inputs)
      loss = loss_function(y_hat, targets)
      loss.backward()
      optimizer.step()
      total_loss += loss
    if (verbose):
      print(epoch, total_loss)
```

In [9]:

```
def clustering_get_accuracy(n_clusters, keys, matrix, cluster_method, flag_empty_
```

```
  if (cluster_method == "kmeans"):
    cluster_algo = KMeans(n_clusters)
  elif (cluster_method == "agglomerative"):
      cluster_algo = AgglomerativeClustering(
        n_clusters=n_clusters
      )
  elif (cluster_method == "agglomerative_complete"):
    cluster_algo = AgglomerativeClustering(
      n_clusters=n_clusters,
      linkage="complete",
      affinity="cosine"
    )
  cluster_algo.fit(matrix)
  clusters = []
  for i in range(50):
    clusters.append(set())
  i = 0
  for label in cluster_algo.labels_:
    clusters[label].add(keys[i])
    i += 1
  correct = 0
  for cluster in clusters:
    if (flag_empty_clusters and len(cluster) == 0):
      print("EMPTY CLUSTER DETECTED")
    if (print_cluster):
      print(cluster)
    for word in cluster:
      if word[::-1] in cluster:
        correct += 1
  return correct / len(keys)
```

In [10]:

```
def get_target_words_embeddings(target_words, embedding_matrix, word_to_idx):
  keys = []
  matrix = []
  for key in target_words:
    idx = word_to_idx[key]
    matrix.append(embedding_matrix[idx])
    keys.append(key)
  return (keys, matrix)
```

In [11]:

```
def run_experiment(iterations, corpus_root, training_epochs, embedding_dims, wind
  accuracy = np.zeros(iterations)
  # Get the 50 most common words for the experiment
  processed_corpus = process_corpus(original_corpus, True, True, False, True, 
  most_frequent_tokens = most_frequent(processed_corpus, 50, False)
  cluster_words = set()
  for (word, freq) in most_frequent_tokens:
    cluster_words.add(word)
    cluster_words.add(word[::-1])

  for iteration in range(iterations):
    # reverse half of instances of most common words at random
    generate_corpus_half_tokens_reversed(original_corpus, most_frequent_tokens, T
    # clean sentences
    sentences = get_all_sentences_cleaned(corpus_after_token_reversal, stemming, 

    # set up data
    (word_to_idx, idx_to_word) = generate_word_to_indx_and_idx_to_word(sentences)
    vocab_size = len(word_to_idx)
    tuples = get_context_window_tuples(word_to_idx, sentences, window_size, clust
    (words, contexts) = get_skipgrams(sentences, word_to_idx, window_size, 10)
```

```
    # train model
    skipgrams_model = Word2Vec_Skipgram(embedding_dims, vocab_size=vocab_size).to
    train_skipgram_model(skipgrams_model, training_epochs, 500, learning_rate, Fa

    # get embeddings
    embedding_matrix = skipgrams_model.embedding_words.weight.detach().cpu().num
    (target_words, embeddings) = get_target_words_embeddings(cluster_words, embed

    # perform clustering
    accuracy[iteration] = clustering_get_accuracy(50, target_words, embeddings, c
    if (verbose):
      print("Iteration", iteration + 1, "Accuracy:", accuracy[iteration])
  return ("Average accuracy:", np.mean(accuracy), "Standard diviation:", np.std(
```

In [12]:

```
print("Performance with window size 1:",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20, e
                     cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                     verbose=False))

print("Performance with window size 2:",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20, e
                     cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                     verbose=False))

print("Performance with window size 3:",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20, e
                     cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                     verbose=False))

print("Performance with window size 5:",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20, e
                     cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                     verbose=False))
```

```
Performance with window size 1: ('Average accuracy:', 0.876, 'Standard diviation:'
Performance with window size 2: ('Average accuracy:', 0.8560000000000001, 'Standar
Performance with window size 3: ('Average accuracy:', 0.8400000000000001, 'Standar
Performance with window size 5: ('Average accuracy:', 0.716, 'Standard diviation:'
```

In [13]:

```
print("Performance with word embedding length 50:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
                   cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                   verbose=False))

print("Performance with embedding length 100:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
                   cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                   verbose=False))

print("Performance with embedding length 150:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
                   cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                   verbose=False))

print("Performance with embedding length 200:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
                   cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                   verbose=False))
print("Performance with embedding length 300:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
                   cluster_method="agglomerative_complete", learning_rate=0.01, stemm
                   verbose=False))
```

```
print("Performance with embedding length 400:",
    run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40, emb
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))
```

```
Performance with word embedding length 50: ('Average accuracy:', 0.828000000000000
Performance with embedding length 100: ('Average accuracy:', 0.884, 'Standard divi
Performance with embedding length 150: ('Average accuracy:', 0.884, 'Standard divi
Performance with embedding length 200: ('Average accuracy:', 0.9, 'Standard diviat
Performance with embedding length 300: ('Average accuracy:', 0.9, 'Standard diviat
Performance with embedding length 400: ('Average accuracy:', 0.8799999999999999, '
```

In [14]:

```
 print("Stemming: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))

print("Performance with stopwords removal:",
     run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20, e
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))

print("Baseline: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))
```

```
Stemming:  ('Average accuracy:', 0.9079999999999998, 'Standard diviation:', 0.0324
Performance with stopwords removal: ('Average accuracy:', 0.7799999999999999, 'Sta
Baseline:  ('Average accuracy:', 0.924, 'Standard diviation:', 0.0319999999999998
```

In [15]:

```
print("10 Epochs: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=10,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))

print("20 Epochs: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=20,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))

print("30 Epochs: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=30,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))

print("40 Epochs: ",
      run_experiment(iterations=5, corpus_root=corpus_root, training_epochs=40,
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=False))
```

```
10 Epochs:  ('Average accuracy:', 0.916, 'Standard diviation:', 0.0195959179422654
20 Epochs:  ('Average accuracy:', 0.9199999999999999, 'Standard diviation:', 0.012
30 Epochs:  ('Average accuracy:', 0.8959999999999999, 'Standard diviation:', 0.034
40 Epochs:  ('Average accuracy:', 0.876, 'Standard diviation:', 0.0427083130081252
```

In [17]:

```
 print("Best performance: ",
      run_experiment(iterations = 5, corpus_root=corpus_root, training_epochs=2(
            cluster_method="agglomerative_complete", learning_rate=0.01, stemr
            verbose=True))
```

```
{'serutaef'. 'features'}
```

```
{'teg', 'get'}
{'need', 'want'}
{'erawtfos', 'using'}
{'doog', 'taerg', 'good', 'great'}
{'orcim', 'creative', 'evitaerc', 'micro'}
{'elttil', 'little'}
{'osla', 'also'}
{'desu', 'used'}
{'tsrif', 'first'}
{'product', 'tcudorp'}
{'even', 'neve'}
{'software', 'nez', 'zen'}
{'computer', 'retupmoc'}
{'enohp', 'phone'}
{'dnuos', 'sound'}
{'yllaer', 'really'}
{'reyalp', 'player'}
{'ipod', 'dopi'}
{'3pm', 'mp3'}
{'would', 'dluow'}
{'deen', 'tnaw'}
{'hcum', 'much'}
{'price', 'ecirp'}
{'llew', 'well'}
{'serutcip', 'pictures'}
{'ezis', 'size'}
{'one', 'eno'}
{'retuor', 'router'}
{'notron', 'norton'}
{'life', 'efil'}
{'use', 'esu'}
{'time', 'emit'}
{'like'}
{'works', 'skrow'}
{'yub', 'buy'}
{'retteb', 'better'}
{'champ', 'pmahc'}
{'ysae', 'easy'}
{'ytilauq', 'quality'}
{'ekil'}
{'cisum', 'music'}
{'still'}
{'llits'}
{'battery', 'yrettab'}
{'melborp', 'problem'}
{'camera', 'aremac'}
{'krow', 'work'}
{'gnisu'}
{'repaid', 'diaper'}
Iteration 1 Accuracy: 0.88
{'nez', 'reyalp', 'player', 'zen'}
{'doog', 'taerg', 'good', 'great'}
{'even', 'neve'}
{'teg', 'get'}
{'still', 'llits'}
{'use', 'esu'}
{'serutcip', 'pictures'}
{'ezis', 'size'}
{'want', 'tnaw'}
{'champ', 'pmahc'}
{'osla', 'also'}
{'tsrif', 'first'}
{'llew', 'well'}
{'computer', 'retupmoc'}
```

```
{'retteb', 'better'}
{'price', 'ecirp'}
{'works', 'skrow'}
{'krow', 'work'}
{'time', 'emit'}
{'erawtfos', 'software'}
{'enohp', 'phone'}
{'3pm', 'mp3'}
{'need', 'deen'}
{'retuor', 'router'}
{'dnuos', 'sound'}
{'life', 'efil'}
{'notron', 'norton'}
{'elttil', 'little'}
{'orcim', 'creative', 'evitaerc', 'micro'}
{'desu', 'used'}
{'ekil', 'like'}
{'product'}
{'using'}
{'melborp', 'problem'}
{'hcum', 'much'}
{'ytilauq', 'quality'}
{'one', 'eno'}
{'ipod', 'dopi'}
{'yub', 'buy'}
{'yllaer', 'really'}
{'repaid', 'diaper'}
{'battery', 'yrettab'}
{'tcudorp'}
{'ysae', 'easy'}
{'gnisu'}
{'cisum', 'music'}
{'features'}
{'camera', 'aremac'}
{'would', 'dluow'}
{'serutaef'}
Iteration 2 Accuracy: 0.94
{'orcim', 'creative', 'evitaerc', 'micro'}
{'yub', 'buy'}
{'yllaer', 'really', 'still'}
{'osla', 'also'}
{'teg', 'get'}
{'ezis', 'size'}
{'time', 'emit'}
{'reyalp', 'player'}
{'serutcip', 'pictures'}
{'llew', 'well'}
{'doog', 'taerg', 'good', 'great'}
{'want', 'tnaw'}
{'erawtfos', 'software'}
{'elttil', 'little'}
{'one', 'eno'}
{'even', 'neve'}
{'krow', 'work'}
{'would', 'dluow'}
{'ipod', 'dopi'}
{'desu', 'used'}
{'melborp', 'problem'}
{'ekil', 'like'}
{'tsrif', 'first'}
{'cisum', 'music'}
{'use', 'esu'}
{'serutaef', 'features'}
{'life', 'efil'}
{'product', 'tcudorp'}
```

```
{'product', 'tcudorp'}
{'using'}
{'hcum', 'much'}
{'retuor', 'router'}
{'dnuos', 'sound'}
{'enohp', 'phone'}
{'battery', 'yrettab'}
{'champ', 'pmahc'}
{'llits'}
{'retteb', 'better'}
{'3pm', 'mp3'}
{'repaid', 'diaper'}
{'ytilauq', 'quality'}
{'retupmoc'}
{'works', 'skrow'}
{'notron', 'norton'}
{'price', 'ecirp'}
{'need', 'deen'}
{'camera', 'aremac'}
{'computer'}
{'gnisu'}
{'ysae', 'easy'}
{'nez', 'zen'}
Iteration 3 Accuracy: 0.94
{'ezis', 'battery', 'size', 'yrettab'}
{'computer', 'retupmoc'}
{'creative', 'evitaerc', 'nez', 'zen'}
{'doog', 'taerg', 'good', 'great'}
{'ekil', 'like'}
{'llew', 'well'}
{'osla', 'also'}
{'using', 'gnisu'}
{'need', 'deen', 'tnaw'}
{'hcum', 'much'}
{'tsrif', 'first'}
{'enohp', 'phone'}
{'even', 'neve'}
{'dnuos', 'sound'}
{'reyalp', 'player'}
{'time', 'emit'}
{'still', 'llits'}
{'yub', 'buy'}
{'melborp', 'problem'}
{'notron', 'norton'}
{'teg', 'get'}
{'would', 'dluow'}
{'champ', 'pmahc'}
{'elttil', 'little'}
{'works', 'skrow'}
{'camera', 'aremac'}
{'ytilauq', 'quality'}
{'yllaer', 'really'}
{'erawtfos', 'software'}
{'retteb', 'better'}
{'krow', 'work'}
{'use', 'esu'}
{'retuor', 'router'}
{'desu', 'used'}
{'ipod', 'dopi'}
{'pictures'}
{'life', 'efil'}
{'orcim', 'micro'}
{'price', 'ecirp'}
{'3pm', 'mp3'}
{'product'}
```

```
{'one', 'eno'}
{'features'}
{'repaid', 'diaper'}
{'serutcip'}
{'ysae', 'easy'}
{'want'}
{'tcudorp'}
{'serutaef'}
{'cisum', 'music'}
Iteration 4 Accuracy: 0.92
{'doog', 'taerg', 'good', 'great'}
{'still', 'llits'}
{'orcim', 'creative', 'evitaerc', 'micro'}
{'osla', 'also'}
{'ezis', 'size'}
{'deen', 'need', 'want'}
{'desu', 'used'}
{'yllaer', 'really'}
{'teg', 'get'}
{'would', 'dluow'}
{'works', 'krow', 'skrow'}
{'ekil', 'like'}
{'product', 'tcudorp'}
{'using', 'gnisu'}
{'elttil', 'little'}
{'champ', 'pmahc'}
{'dnuos', 'sound'}
{'retteb', 'better'}
{'one', 'eno'}
{'notron', 'norton'}
{'even', 'neve'}
{'hcum', 'much'}
{'tsrif', 'first'}
{'llew', 'well'}
{'ipod', 'dopi'}
{'price', 'ecirp'}
{'computer', 'retupmoc'}
{'time', 'emit'}
{'serutcip', 'pictures'}
{'retuor', 'router'}
{'enohp', 'phone'}
{'use', 'esu'}
{'life', 'efil'}
{'melborp', 'problem'}
{'erawtfos'}
{'cisum', 'music'}
{'ytilauq', 'quality'}
{'battery', 'yrettab'}
{'yub', 'buy'}
{'features'}
{'work'}
{'3pm', 'mp3'}
{'serutaef'}
{'camera', 'aremac'}
{'nez', 'zen'}
{'reyalp', 'player'}
{'ysae', 'easy'}
{'software'}
{'tnaw'}
{'repaid', 'diaper'}
Iteration 5 Accuracy: 0.92
Best performance:  ('Average accuracy:', 0.9199999999999999, 'Standard diviation:'
```