```
%%bash
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
rm -rf 6864-hw1
git clone https://github.com/lingo-mit/6864-hw1.git
```

```
Cloning into '6864-hw1'...
```

```
import sys
sys.path.append("/content/6864-hw1")

import csv
import itertools as it
import numpy as np
np.random.seed(0)

import lab_util
```

Introduction

In this lab, you'll explore three different ways of using unlabeled text data to learn pretrained word repr the effects of different modeling decisions (representation learning objective, context size, etc.) on bo representations and their effect on a downstream prediction problem.

General lab report guidelines

Homework assignments should be submitted in the form of a research report. (We'll be providing a pl but are still sorting out some logistics.) Please upload PDFs, with a maximum of four single-spaced pa Association for Computational Linguistics style files.) Reports should have one section for each part o section should describe the details of your code implementation, and include whatever charts / tables questions at the end of the corresponding homework part.

We're going to be working with a dataset of product reviews. It looks like this:

```
data = []
n_positive = 0
n_disp = 0
with open("/content/6864-hw1/reviews.csv") as reader:
  csvreader = csv.reader(reader)
  next(csvreader)
  for id, review, label in csvreader:
    label = int(label)

    # hacky class balancing
    if label == 1:
      if n_positive == 2000:
        continue
```

```
      n_positive += 1
    if len(data) == 4000:
      break

    data.append((review, label))

    if n_disp > 5:
      continue
    n_disp += 1
    print("review:", review)
    print("rating:", label, "(good)" if label == 1 else "(bad)")
    print()

print(f"Read {len(data)} total reviews.")
np.random.shuffle(data)
reviews, labels = zip(*data)
train_reviews = reviews[:3000]
train_labels = labels[:3000]
val_reviews = reviews[3000:3500]
val_labels = labels[3000:3500]
test_reviews = reviews[3500:]
test_labels = labels[3500:]
```

```
review: I have bought several of the Vitality canned dog food products and have f
rating: 1 (good)

review: Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actual
rating: 0 (bad)

review: This is a confection that has been around a few centuries.  It is a light
rating: 1 (good)

review: If you are looking for the secret ingredient in Robitussin I believe I ha
rating: 0 (bad)

review: Great taffy at a great price.  There was a wide assortment of yummy taffy
rating: 1 (good)

review: I got a wild hair for taffy and ordered this five pound bag. The taffy wa
rating: 1 (good)

Read 4000 total reviews.
```

We've provided a little bit of helper code for reading in the dataset; your job is to implement the learnir

Part 1: word representations via matrix factorization

First, we'll construct the term--document matrix (look at `/content/6864-hw1/lab_util.py` in the file how this works).

```
vectorizer = lab_util.CountVectorizer()
vectorizer.fit(train_reviews)
td_matrix = vectorizer.transform(train_reviews).T
print(f"TD matrix is {td_matrix.shape[0]} x {td_matrix.shape[1]}")
```

```
TD matrix is 2006 x 3000
```

First, implement a function that computes word representations via latent semantic analysis:

```
def learn_reps_lsa(matrix, rep_size):
  # `matrix` is a `|V| x n` matrix, where `|V|` is the number of words in the
  # vocabulary. This function should return a `|V| x rep_size` matrix with each
  # row corresponding to a word representation. The `sklearn.decomposition`
  # package may be useful.

  import sklearn.decomposition
  solver = sklearn.decomposition.TruncatedSVD(n_components=rep_size)
  u = solver.fit_transform(matrix)
  return u
```

Let's look at some representations:

```
reps = learn_reps_lsa(td_matrix, 64)
words = ["good", "bad", "cookie", "jelly", "dog", "the", "4"]
show_tokens = [vectorizer.tokenizer.word_to_token[word] for word in words]
lab_util.show_similar_words(vectorizer.tokenizer, reps, show_tokens)
```

```
good 47
  pretty 0.645
  really 0.830
  everyone 0.838
  liked 0.846
  better 0.861
bad 201
  ok 0.589
  taste 0.593
  either 0.607
  really 0.628
  . 0.648
cookie 504
  nana's 0.466
  cookies 0.602
  shortbread 0.794
  hope 0.817
  gluten 0.870
jelly 351
  online 1.023
  low 1.038
  hoping 1.059
  look 1.063
  twist 1.103
dog 925
  pet 0.264
  dogs 0.307
  food 0.370
  nutritious 0.407
  pets 0.422
the 36
  . 0.331
  <unk> 0.366
  of 0.394
  and 0.402
  to 0.422
4 292
  1 0.196
  6 0.199
  2 0.317
  70 0.426
  5 0.497
```

We've been operating on the raw count matrix, but in class we discussed several reweighting schemes more informative.

Here, implement the TF-IDF transform and see how it affects learned representations.

```
def transform_tfidf(matrix):
    # `matrix` is a `|V| x |D|` matrix of raw counts, where `|V|` is the
    # vocabulary size and `|D|` is the number of documents in the corpus. This
    # function should (nondestructively) return a version of `matrix` with the
    # TF-IDF transform appliied.
```

```
    thresholded = matrix > 1
    dfs = thresholded.sum(axis=1)[:, np.newaxis]
    idfs = np.log(matrix.shape[1]) - np.log(dfs + 1e-8)
    return matrix * idfs
```

How does this change the learned similarity function?

```
td_matrix_tfidf = transform_tfidf(td_matrix)
reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 64)
lab_util.show_similar_words(vectorizer.tokenizer, reps_tfidf, show_tokens)
```

```
good 47
  but 0.334
  . 0.397
  is 0.441
  as 0.445
  too 0.457
bad 201
  taste 0.373
  like 0.434
  but 0.512
  not 0.537
  just 0.566
cookie 504
  nana's 0.627
  cookies 0.627
  gluten 0.792
  flour 0.828
  free 0.855
jelly 351
  mixing 0.958
  save 0.986
  gifts 0.990
  creamer 1.020
  okay 1.025
dog 925
  pet 0.327
  dogs 0.410
  food 0.476
  switched 0.538
  pets 0.546
the 36
  . 0.084
  of 0.106
  to 0.119
  and 0.133
  in 0.154
4 292
  6 0.169
  1 0.197
  2 0.469
  70 0.528
  5 0.592
```

Now that we have some representations, let's see if we can do something useful with them.

Below, implement a feature function that represents a document as the sum of its learned word embe

The remaining code trains a logistic regression model on a set of *labeled* reviews; we're interested in s from *unlabeled* reviews improve classification.

```
REP_DICT = learn_reps_lsa(td_matrix_tfidf, 64)

def word_featurizer(xs):
  # normalize
  return xs / np.sqrt((xs ** 2).sum(axis=1, keepdims=True))

def lsa_featurizer(xs):
  # This function takes in a matrix in which each row contains the word counts
  # for the given review. It should return a matrix in which each row contains
  # the learned feature representation of each review (e.g. the sum of LSA
  # word representations).

  feats = sum(np.outer(xs[:, i], REP_DICT[i, :]) for i in range(xs.shape[1]))
  return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

def combo_featurizer(xs):
  return np.concatenate((word_featurizer(xs), lsa_featurizer(xs)), axis=1)

def train_model(featurizer, xs, ys):
  import sklearn.linear_model
  xs_featurized = featurizer(xs)
  model = sklearn.linear_model.LogisticRegression()
  model.fit(xs_featurized, ys)
  return model

def eval_model(model, featurizer, xs, ys):
  xs_featurized = featurizer(xs)
  pred_ys = model.predict(xs_featurized)
  print("test accuracy", np.mean(pred_ys == ys))

def training_experiment(name, featurizer, n_train):
  print(f"{name} features, {n_train} examples")
  train_xs = vectorizer.transform(train_reviews[:n_train])
  train_ys = train_labels[:n_train]
  test_xs = vectorizer.transform(test_reviews)
  test_ys = test_labels
  model = train_model(featurizer, train_xs, train_ys)
  eval_model(model, featurizer, test_xs, test_ys)
  print()

training_experiment("word", word_featurizer, 20)
training_experiment("lsa", lsa_featurizer, 20)
training_experiment("combo", combo_featurizer, 20)
```

```
training_experiment("word", word_featurizer, 100)
training_experiment("lsa", lsa_featurizer, 100)
training_experiment("combo", combo_featurizer, 100)

training_experiment("word", word_featurizer, 1000)
training_experiment("lsa", lsa_featurizer, 1000)
training_experiment("combo", combo_featurizer, 1000)

training_experiment("word", word_featurizer, 3000)
training_experiment("lsa", lsa_featurizer, 3000)
training_experiment("combo", combo_featurizer, 3000)
```

```
word features, 20 examples
test accuracy 0.526

lsa features, 20 examples
test accuracy 0.524

combo features, 20 examples
test accuracy 0.526

word features, 100 examples
test accuracy 0.616

lsa features, 100 examples
test accuracy 0.6

combo features, 100 examples
test accuracy 0.628

word features, 1000 examples
test accuracy 0.784

lsa features, 1000 examples
test accuracy 0.682

combo features, 1000 examples
test accuracy 0.782

word features, 3000 examples
test accuracy 0.784

lsa features, 3000 examples
test accuracy 0.734

combo features, 3000 examples
test accuracy 0.796
```

Part 1: Lab writeup

Part 1 of your lab report should discuss any implementation details that were important to filling out t up experiments that answer the following questions:

1. Qualitatively, what do you observe about nearest neighbors in repres words are most similar to *the*, *dog*, *3*, and *good*?)

For both LSA and TFIDF, with rep size 500, the nearest neighbors of for `good` and `bad` consist of com punctuation. It does not seem to have learned sentiment association based on the nearest neighbors, similar to `the`.

LSA learned good representations for nouns (`dog` \rightarrow `food`, `pet`, `pets`; `cookie` -> `nana's`, are some combination of synonyms and closely associated words.

TFIDF noun representations did not have as good nearest neighbors (`cookie` -> `walmart`, `effort`, `c` `cardboard`, `advertised`).

Both LSA and TFIDF learned to associate numbers with other numbers.

2. How does the size of the LSA representation affect this behavior?

Reducing the LSA representation to 10 led to nearest neighbors with higher variance in semantic mea didn't make sense at all. This behavior is expected because it is harder to encode semantic similarity w

LSA rep size 10 examples:

- `good` -> `better`, `like`, `good`
- `bad` -> `taste`, `like`, `better`
- `dog` -> `dogs`, `vet`, `food`
- `jelly` -> `freezer`, `ate`, `unable`
- `4` -> `1`, `2`, `br`

TFIDF rep size 10 examples:

- `good` -> `like`, `too`, `come`
- `bad` -> `me`, `nor`, `lobster`
- `dog` -> `dogs`, `we`, `pet`
- `jelly` -> `mess`, `touch`, `needed`
- `4` -> `6`, `1`, `11`

3. Recall that the we can compute the word co-occurrence matrix W_{tt} prove about the relationship between the left singular vectors of W_{td} a behavior with your implementation of `learn_reps_lsa`? Why or why n

4. Do learned representations help with the review classification proble between the number of labeled examples and the effect of word embe

Training results, rep size 64:

- 20 examples
    - word features: **0.526**
    - lsa features: **0.524**
    - combo features: **0.526**
- 100 examples
    - word features: **0.616**
    - lsa features: **0.604**
    - combo features: **0.626**
- 1000 examples
    - word features: **0.784**
    - lsa features: **0.678**
    - combo features: **0.784**
- 3000 examples
    - word features: **0.784**
    - lsa features: **0.738**
    - combo features: **0.794**

In general, the word features performed better than lsa, and about the same as the combination of bo convey a strong sense of sentiment that is important to classifying the a sentence as a whole.

Performance also substantially increased as we increased the number of training examples. This tells good at encoding sentiment or other features specific to this task. This is consistent with the nearest `bad`.

5. What is the relationship between the size of the word embeddings a classification task.

Classification accuracy, 3000 training examples:

- rep size 2:
    - lsa features: **0.544**
    - combo features: **0.784**
- rep size 4:
    - lsa features: **0.610**
    - combo features: **0.790**
- rep size 8:

  - lsa features: **0.612**
  - combo features: **0.788**
- rep size 16:
  - lsa features: **0.644**
  - combo features: **0.794**
- rep size 16:
  - lsa features: **0.686**
  - combo features: **0.796**
- rep size 32:
  - lsa features: **0.738**
  - combo features: **0.794**

There is a trend towards increasing accuracy with larger representation size. However, after a certain this example). We could infer that the model begins to overfit when the representation size is too larg features to perform well when the representation small is too small.

Part 2: word representations via language modeling

In this section, we'll train a word embedding model with a word2vec-style objective rather than a matri little more work; we've provided scaffolding for a PyTorch model implementation below. (If you've nev tutorials [here](). You're also welcome to implement these experiments in any other framework of your cl

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as torch_data

class Word2VecModel(nn.Module):
  # A torch module implementing a word2vec predictor. The `forward` function
  # should take a batch of context word ids as input and predict the word
  # in the middle of the context as output, as in the CBOW model from lecture.

  def __init__(self, vocab_size, embed_dim):
      super().__init__()

      self.embeds = nn.Embedding(vocab_size, embed_dim)
      self.linear1 = nn.Linear(embed_dim, 64)
      self.linear2 = nn.Linear(64, vocab_size)

  def forward(self, context):
      # Context is an `n_batch x n_context` matrix of integer word ids
```

```
        # this function should return a set of scores for predicting the word
        # in the middle of the context

        output = self.embeds(context)  # get the embeddings
        output = output.sum(dim=1)
        output = F.relu(self.linear1(output))  # pass through first layer
        output = self.linear2(output)  # pass through second layer
        return output


import time

def learn_reps_word2vec(corpus, window_size, rep_size, n_epochs, n_batch):
  # This method takes in a corpus of training sentences. It returns a matrix of
  # word embeddings with the same structure as used in the previous section of
  # the assignment. (You can extract this matrix from the parameters of the
  # Word2VecModel.)

  tokenizer = lab_util.Tokenizer()
  tokenizer.fit(corpus)
  tokenized_corpus = tokenizer.tokenize(corpus)

  ngrams = lab_util.get_ngrams(tokenized_corpus, window_size)

  device = torch.device('cuda')  # run on colab gpu
  model = Word2VecModel(tokenizer.vocab_size, rep_size).to(device)
  opt = optim.Adam(model.parameters(), lr=0.001)
  loss_fn = nn.CrossEntropyLoss()

  loader = torch_data.DataLoader(ngrams, batch_size=n_batch, shuffle=True)

  start = time.time()
  for epoch in range(n_epochs):
    epoch_loss = 0
    n_batches = 0
    for context, label in loader:
      # as described above, `context` is a batch of context word ids, and
      # `label` is a batch of predicted word labels

      preds = model(context.to(device))
      loss = loss_fn(preds, label.to(device))

      opt.zero_grad()
      loss.backward()
      opt.step()

      epoch_loss += loss.item()
      n_batches += 1

    epoch_loss /= n_batches
    if (epoch+1)%10 == 0:
      print(f"epoch {epoch+1}: {epoch_loss}, {time.time()-start}s")
```

```
    start = time.time()

  # reminder: you want to return a `vocab_size x embedding_size` numpy array
  embedding_matrix = next(model.embeds.parameters())
  return embedding_matrix.cpu().detach().numpy()
```

```
reps_word2vec = learn_reps_word2vec(train_reviews, 1, 32, 200, 1024)
```

```
epoch 10: 4.256700931863392, 17.623494386672974s
epoch 20: 3.983307274093342, 17.41620683670044s
epoch 30: 3.8509539995300637, 17.40672516822815s
epoch 40: 3.768364041039113, 17.36606478691101s
epoch 50: 3.7114413472150596, 17.176886320114136s
epoch 60: 3.6672060105684543, 16.935622453689575s
epoch 70: 3.6327227724625377, 17.267109394073486s
epoch 80: 3.6048706292213124, 16.827821731567383s
epoch 90: 3.5812128411696644, 17.252073764801025s
epoch 100: 3.561018699117368, 16.78904128074646s
epoch 110: 3.5436181236295665, 17.106294870376587s
epoch 120: 3.528318163160974, 17.02020502090454s
epoch 130: 3.5150228912910717, 17.30059576034546s
epoch 140: 3.5028486992982444, 17.06654977798462s
epoch 150: 3.4929499233260137, 17.280428647994995s
epoch 160: 3.4833158285876786, 17.535560369491577s
epoch 170: 3.4741478278842313, 17.21116876602173s
epoch 180: 3.4660481370790173, 17.335447311401367s
epoch 190: 3.459195370084784, 17.130173206329346s
epoch 200: 3.4520903330170705, 17.354069471359253s
```

After training the embeddings, we can try to visualize the embedding space to see if it makes sense. F
and check its closest neighbors.

```
lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, show_tokens)
```

```
good 47
  great 0.577
  bad 0.606
  awesome 0.613
  decent 0.732
  funny 0.802
bad 201
  good 0.606
  funny 0.734
  great 0.752
  wonderful 0.791
  pleasant 0.831
cookie 504
  natural 0.893
  marinade 0.974
  enough 1.029
  cherry 1.068
  basil 1.102
jelly 351
  breed 0.944
  flakes 1.008
  sandwiches 1.042
  sticks 1.046
  pods 1.046
dog 925
  cat 0.426
  baby 0.663
  junk 0.922
  breath 0.947
  current 0.975
the 36
  their 0.620
  my 0.792
  our 0.798
  any 0.850
  your 0.869
4 292
  10 0.505
  6 0.567
  3 0.629
  5 0.653
  2 0.670
```

We can also cluster the embedding space. Clustering in 4 or more dimensions is hard to visualize, and because there are so many words in the vocabulary. One thing we can try to do is assign cluster labels pattern in the clusters.

```
from sklearn.cluster import KMeans

indices = KMeans(n_clusters=10).fit_predict(reps_word2vec)
zipped = list(zip(range(vectorizer.tokenizer.vocab_size), indices))
np.random.shuffle(zipped)
zipped = zipped[:100]
```

```
zipped = sorted(zipped, key=lambda x: x[1])
for token, cluster_idx in zipped:
  word = vectorizer.tokenizer.token_to_word[token]
  print(f"{word}: {cluster_idx}")
```

```
having: 4
generally: 4
taste: 4
she: 4
zero: 4
son: 4
case: 4
sad: 4
starbucks: 5
book: 5
satisfied: 5
artificial: 5
babies: 5
pain: 5
cravings: 5
others: 5
medium: 5
varieties: 5
opened: 5
white: 5
grounds: 5
blue: 5
carbonated: 6
own: 6
container: 6
summer: 6
claim: 6
spots: 6
word: 6
juice: 6
based: 6
ask: 6
package: 6
although: 6
current: 7
mistake: 7
rice: 7
gold: 7
name: 7
flower: 7
baking: 8
follow: 8
giving: 8
speak: 8
see: 8
leave: 8
consider: 8
have: 8
ship: 8
make: 8
fall: 9
grown: 9
face: 9
animal: 9
shop: 9
missing: 9
brand: 9
```

```
    let: 9
```

Finally, we can use the trained word embeddings to construct vector representations of full reviews. O
average all the word embeddings in the review to create an overall embedding. Implement the transfo
this.

```
def lsa_featurizer(xs):
  feats = sum(np.outer(xs[:, i], reps_word2vec[i, :]) for i in range(xs.shape[1]))

  # normalize
  return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

training_experiment("word2vec", lsa_featurizer, 10)
```

```
training_experiment("word2vec", lsa_featurizer, 10)
training_experiment("word2vec", lsa_featurizer, 100)
training_experiment("word2vec", lsa_featurizer, 1000)
training_experiment("word2vec", lsa_featurizer, 3000)
```

```
word2vec features, 10 examples
test accuracy 0.534

word2vec features, 100 examples
test accuracy 0.608

word2vec features, 1000 examples
test accuracy 0.694

word2vec features, 3000 examples
test accuracy 0.698
```

Part 2: Lab writeup

Part 2 of your lab report should discuss any implementation details that were important to filling out t up experiments that answer the following questions:

1. Qualitatively, what do you observe about nearest neighbors in repres words are most similar to *the*, *dog*, *3*, and *good*?) How well do word2ve to your intuitions about word similarity?

Nearest neighbors for context size 2, embed dimension 32, train epochs 300, batch size 1024, hidden

- `good`
    1. `great` 0.329
    2. `decent` 0.510
    3. `awesome` 0.606
    4. `healthy` 0.611
    5. `bad` 0.617
- `bad`
    1. `good` 0.617
    2. `bitter` 0.619
    3. `supposed` 0.710
    4. `overwhelming` 0.724
    5. `overpowering` 0.729
- `cookie`
    1. `inside` 0.920
    2. `frosting` 0.964
    3. `square` 0.996

4. `berry` 1.005
5. `mug` 1.008

- `jelly`
  1. `peas` 0.928
  2. `earth's` 0.980
  3. `grape` 0.999
  4. `were` 1.003
  5. `kinds` 1.016
- `dog`
  1. `cat` 0.497
  2. `cats` 0.779
  3. `baby` 0.840
  4. `life` 0.858
  5. `lives` 0.861
- `the`
  1. `their` 0.765
  2. `an` 0.792
  3. `your` 0.885
  4. `amazon's` 0.899
  5. `my` 1.026
- `4`
  1. `2` 0.359
  2. `6` 0.417
  3. `3` 0.525
  4. `5` 0.542
  5. `24` 0.600

Most of the nearest neighbors match what I would intuitively expect to see. An interesting observatior
that arise are also dependent on the source word. This indicates that different types of relationships a

- For `good` and `bad`, we get words that are synonyms (ex. `good` -> `great`), words that are antony
  and food related terms that also convey sentiment (ex. `good` -> `healthy` and `bad` -> `bitter`).
- For `cookie`, we get food items that are related to `cookie` but not necessarily synonyms or antc
- `jelly` is a less frequent word in the dataset so its nearest neighbors make the least amount of
  is a neighbor that makes sense as describing a type of jelly.
- `dog` neighbors consist of other living beings (ex. `cat`, `cats`, `baby`) and also literally `life` and
- `the` is interesting because its nearest neighbors are other "structural" words such as articles an
- The number `4` simply gives neighbors that are other numbers.

When we look at the clusters of words generated by the clustering algorithm, we can identify more pa
with our observations above.

0. `right`, `cilantro`, `days`, `date`, `times`, `sale`, `chemicals`, `review`, `meals`, `mixes`
    - words related to time, others
1. `once`, `maybe`, `these`, `absolutely`, `table`, `wine`, `way`, `nutrition`, `sorry`, `kettle`, `obviously`, `worse`,
    - words related to containers, also adverbs
2. `pineapple`, `peanuts`, `jar`, `stevia`, `donut`, `country`
    - pattern unclear but some words are foods
3. `appeal`, `buy`, `happen`, `consume`, `feed`, `work`, `use`, `serve`, `open`, `bring`, `brought`
    - present tense verbs
4. `tiny`, `40`, `95`, `half`, `rica`, `spoon`, `square`, `disposable`, `harmony`, `several`, `couple`, `reduced`, `lb`, `va`
    - units of measurement
5. `goes`, `has`, `i'll`, `into`, `eats`, `needs`, `wouldn't`, `you'd`, `we'll`
    - some verbs of desire or intent
6. `figured`, `iron`, `decided`, `learned`, `served`, `thrilled`, `concerned`
    - past tense verbs
7. `carbohydrate`, `body`, `gluten`, `coke`, `adult`, `eggs`, `fat`
    - nutrition related words
8. `strange`, `bad`, `perfectly`, `german`, `healthier`, `quite`, `fresh`, `thick`, `incredible`, `pretty`, `range`, `w`
    - adjectives
9. `lemon`, `cake`, `salads`, `canned`, `150`, `flavored`, `benefits`
    - pattern unclear

2. One important parameter in word2vec-style models is context size. H context size affect the kinds of representations that are learned?

I ran the training using context sizes of 1 and 6.

For commonly occurring words with local semantic relationships (ex. adjectives such as `good` and `b`
representations were mostly similar to the ones produced by context size 2 above. It makes sense tha
representation with context size 1, because they often describe the noun that directly follows.

For less commonly occurring words with medium distance semantic relationships (ex. nouns such as
representations made less sense under both context sizes 1 and 6.

In context size 1, `cookie` -> `alive`, `green`, `count`; `jelly` -> `homemade`, `process`, `spots`; `dog` -> `s
associated as direct/indirect objects relative to verbs, and many times the corresponding verb is not c
context size 1, it makes sense that the representation of nouns is lower quality.

In context size 6, `cookie` -> `cereal`, `liquid`, `http`; `jelly` -> `soaked`, `grinder`, `house`; `dog` -> `ca
contributing to the representation of the noun in this case, which is often greater than the length of th
makes sense that the nearest neighbors are other nouns that are not closely related to the original no

3. How do results on the downstream classification problem compare

word2vec features

- 10 examples: **0.498**
- 100 examples: **0.588**
- 1000 examples: **0.716**
- 3000 examples: **0.726**

The word2vec results are slightly worse, which was initially surprising because I found the embedding
accurate than in part 1. One explanation for this result is that the word2vec embeddings model a lot o
measurement, parts of speech, etc.) that are not important to the sentiment classification task.

4. What are some advantages and disadvantages of learned embeddin the featurization done in part 1?

Advantages: the learned embeddings capture more complicated trends in the embedding space, as sh
nearest neighbors produced by word2vec have much better semantic similarity. Representations of in
surrounding context words during training.

Disadvantages: it is hard to get interpretability in the word2vec model. Why is `their` closer to `the` th
components of the embedding space represent? The embedding space is also created in a non deter
established between words in one session of training may be different from another. These inconsist
downstream tasks, especially because the high dimensionality of the embedding space is hard to visu

5. What are some potential problems with constructing a representatio the embeddings of the individual words?

There are a few potential problems.

1. We make the assumption that each word should be given an equal weight when contributing to the overall ser
occurring words dominating sentence representations, despite having low importance.
2. We lose the grammatical structure / ordering of words in the sentence. This could be bad if sentence structur
sentiment (ex. if poor structure indicates negative sentiment).