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

```
Cloning into 'hw1'...
```

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

import csv
import itertools as it
import numpy as np
import sklearn.decomposition
np.random.seed(0)
from tqdm import tqdm

import lab_util
```

Introduction

In this notebook, you will find code scaffolding for the word representation parts of Homework 1 (matrix factorization and Word2Vec-style language modeling; code for the HMM section of the assignment is released in another notebook). There are certain parts of the scaffolding marked with `# Your code here!` comments where you can fill in code to perform the specified tasks. After implementing the methods in this notebook, you will need to design and perform experiments to evaluate each method and respond to the questions in the Homework 1 handout (available on Canvas). You should be able to complete this assignment without changing any of the scaffolding code, just writing code to fill in the scaffolding and run experiments.

Dataset

We're going to be working with a dataset of product reviews. The following cell loads the dataset and splits it into training, validation, and test sets.

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

```
    # hacky class balancing, trick in order to have a balaced class
    if label == 1:
      if n_positive == 2000:
        continue
      n_positive += 1
    if len(data) == 4000:
      break

    data.append((review, label))

    if n_disp > 5:  # what is the purpose of this line ?, ok to not print too many things
      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 found t
rating: 1 (good)

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

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

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

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

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

Read 4000 total reviews.
```

Part 1: word representations via matrix factorization

First, we'll construct the term-document matrix (look at `/content/hw1/lab_util.py` in the file browser on the left if you want to see 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 the function `learn_reps_lsa` that computes word representations via latent semantic analysis. The `sklearn.decomposition` or `np.linalg` packages may be useful.

```
import sklearn.decomposition as decomposition
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.

    # Your code here!
    truncated_svd = decomposition.TruncatedSVD(n_components=rep_size)
    lsa = truncated_svd.fit_transform(matrix)
    return lsa
```

Sanity check 1

The following cell contains a simple sanity check for your `learn_reps_lsa` implementation: it should print `True` if your `learn_reps_lsa` function is implemented equivalently to one of our solutions. There are at least two reasonable ways to formulate these LSA word representations (whether you directly use the left singular vectors of `matrix` or scale them by the singular values), these correspond to the two possible representations in the sanity check below.

```
DEBUG_sc1_matrix = np.array([[1,0,0,2,1,3,5],
                             [2,0,0,0,0,4,0],
                             [0,3,4,1,8,6,6],
                             [1,4,5,0,0,0,0]])

DEBUG_reps = learn_reps_lsa(DEBUG_sc1_matrix, 3)
eigen = np.sqrt(sorted(np.linalg.eigvals(DEBUG_sc1_matrix@DEBUG_sc1_matrix.T))[1:][::-1])
DEBUG_gt1 = np.array([[ -4.92017554,  -2.85465774,   1.18575453],
                      [ -2.14977584,  -1.19987977,   3.37221899],
                      [-12.62664695,   0.10890093,  -1.32131745],
                      [ -2.69216011,   5.66453534,   1.33728063]])
print(eigen)
DEBUG_gt2 = np.array([[-0.35188159, -0.44213061,  0.29358929],
```

```
                  [-0.15374788, -0.18583789,  0.83495136],
                  [-0.90303377,  0.01686662, -0.32715426],
                  [-0.19253817,  0.87732566,  0.3311067 ]])
print(DEBUG_gt1/DEBUG_gt2)
print(np.allclose(np.abs(DEBUG_reps), np.abs(DEBUG_gt1)) or np.allclose(np.abs(DEBUG_reps), r
```

```
[13.98247485  6.45659371  4.03882087]
[[13.98247501  6.45659377  4.0388208 ]
 [13.98247469  6.45659381  4.03882088]
 [13.98247482  6.45659474  4.03882086]
 [13.9824748   6.45659371  4.03882081]]
True
```

Let's look at some representations:

```
reps = learn_reps_lsa(td_matrix, 500)
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
  . 1.056
  a 1.101
  but 1.121
  , 1.152
  the 1.157
bad 201
  . 1.396
  taste 1.416
  but 1.434
  a 1.435
  i 1.449
cookie 504
  nana's 0.777
  cookies 1.036
  oreos 1.287
  bars 1.362
  bites 1.425
jelly 351
  twist 1.144
  cardboard 1.230
  advertised 1.382
  peanuts 1.406
  plastic 1.454
dog 925
  food 1.048
  pet 1.069
  pets 1.071
  switched 1.208
  foods 1.230
the 36
  . 0.331
  <unk> 0.366
  of 0.395
```

```
      and 0.403
      to 0.422
    4 292
      1 1.046
      6 1.119
      70 1.135
      stevia 1.193
      concentrated 1.247
```

We've been operating on the raw count matrix, but in class we discussed several reweighting schemes aimed at making LSA representations 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.

    # Your code here!
    """This function applies the tf-idf transformation on a term-document matrix"""
    tf = matrix
    D = matrix.shape[1]
    occurences_doc = np.sum(matrix > 0, axis=1)
    idf = np.log(D/occurences_doc)
    return (tf.T*idf).T
```

Sanity check 2

The following cell should print `True` if your `transform_tfidf` function is implemented properly. (*Hint: in our implementation, we use the natural logarithm (base e) when computing inverse document frequency.*)

```
DEBUG_sc2_matrix = np.array([[3,1,0,3,0],
                             [0,2,0,0,1],
                             [7,8,2,0,1],
                             [1,9,8,1,0]])
DEBUG_gt = np.array([[1.53247687, 0.51082562, 0.        , 1.53247687, 0.        ],
                     [0.        , 1.83258146, 0.        , 0.        , 0.91629073],
                     [1.56200486, 1.78514841, 0.4462871 , 0.        , 0.22314355],
                     [0.22314355, 2.00829196, 1.78514841, 0.22314355, 0.        ]])
print(np.allclose(transform_tfidf(DEBUG_sc2_matrix), DEBUG_gt))
```

```
    True
```

How does this change the learned similarity function?

```
td_matrix_tfidf = transform_tfidf(td_matrix)
reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 500)
# reps_tfidf = learn_reps_lsa(td_matrix_tfidf, 100)
lab_util.show_similar_words(vectorizer.tokenizer, reps_tfidf, show_tokens)
```

```
good 47
  . 0.980
  but 1.014
  a 1.032
  and 1.086
  is 1.091
bad 201
  . 1.330
  taste 1.339
  but 1.355
  a 1.371
  not 1.381
cookie 504
  nana's 0.810
  cookies 1.159
  bars 1.435
  bites 1.449
  moist 1.452
jelly 351
  twist 1.088
  cardboard 1.230
  advertised 1.361
  plum 1.493
  sold 1.538
dog 925
  food 1.031
  pets 1.096
  pet 1.102
  foods 1.186
  switched 1.255
the 36
  . 0.212
  and 0.270
  <unk> 0.292
  of 0.300
  to 0.322
4 292
  1 0.988
  6 1.052
  70 1.151
  stevia 1.174
  3 1.258
```

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 embeddings.

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

```
import sklearn.linear_model

td_matrix_tfidf = transform_tfidf(td_matrix)  # look-up table for the training embeddings

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

def lsa_featurizer(xs, dims=1000):
    # 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).
    features = learn_reps_lsa(td_matrix_tfidf, dims)
    # now, inside features [in rows] we have the embeddings for every word
    resulting_embeddings = []
    for i, review in enumerate(xs):
      review_embeddings = []
      for j, word in enumerate(review):
        if word > 0: # word, with token j is present in review i
          review_embeddings.append(features[j])
      resulting_embeddings.append(np.mean(review_embeddings, axis=0))
    feats = np.array(resulting_embeddings)
    # normalize
    return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

# We've implemented the remainder of the training and evaluation pipeline,
# so you likely won't need to modify the following four functions.
def combo_featurizer(xs):
    return np.concatenate((word_featurizer(xs), lsa_featurizer(xs)), axis=1)

def train_model(featurizer, xs, ys):
    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)
    return np.mean(pred_ys == ys)

def training_experiment_1(name, featurizer, n_train):
    print(f"{name} features, {n_train} examples")
    # print(train_reviews[0])
    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)
    acc = eval_model(model, featurizer, test_xs, test_ys)
    print(acc, '\n')
    return acc

# The following four lines will run a training experiment with all 3k examples
# in training set for each feature type. `training_experiment` may be useful to
# you when performing experiments to answer questions in Part 1 of the Homework
# 1 handout.
n_train = 3000
training_experiment_1("word", word_featurizer, n_train)
training_experiment_1("lsa", lsa_featurizer, n_train)
training_experiment_1("combo", combo_featurizer, n_train)
print()
```

```
word features, 3000 examples
0.784

lsa features, 3000 examples
0.786

combo features, 3000 examples
0.802
```

Part 1: Lab writeup

Part 1 of your lab report should discuss any implementation details that were important to filling out the code above, as well as your answers to the questions in Part 1 of the Homework 1 handout. Below, you can set up and perform experiments that answer these questions (include figures, plots, and tables in your write-up as you see fit).

Experiments for Part 1

1. Relation between the singular vectors of the term-document matrix and the word co-occurence matrix

Let us write the SVD for W_{tt} and W_{td}.

$$W_{tt} = U_{tt}\Sigma_{tt}V_{tt}^T$$
$$W_{td} = U_{td}\Sigma_{td}V_{td}^T$$

Furthermore, we have $W_{tt} = W_{td}W_{td}^T$. And

$$W_{td}W_{td}^T = U_{td}\Sigma_{td}V_{td}^T V_{td}\Sigma_{td}^T U_{td}^T$$

Now, we can usee the identity $V_{td}^T V_{td} = Id$ and get

$$W_{tt} = U_{td}\Sigma_{td}\Sigma_{td}^T U_{td}^T$$

Now, since W_{tt} is diagonalizable and $\Sigma_{td}\Sigma_{td}^T$ is diagonal, we can identify the decomposition and identify that the left singular vectors of W_{tt} and W_{td} are identical.

```
# Your code here!
Wtd = td_matrix
Wtt = Wtd@Wtd.T
eigenvalues = sorted(np.linalg.eigvals(Wtt))[1:][::-1]
truncated_svd = decomposition.TruncatedSVD(n_components=Wtd.shape[0]-1)
UttSigma = truncated_svd.fit_transform(Wtt)/np.sqrt(eigenvalues)
UtdSigma = truncated_svd.fit_transform(Wtd)


ratio = UttSigma/UtdSigma
print(ratio[:, 0])
```

```
[1. 1. 1. ... 1. 1. 1.]
```

Now that we have verified this relation, how could this be useful ? THe co-occurence matrix is helpful for PMI normalization whereas the term-document matrix is useful for TF-IDF normalization. Once these operations are being done on top of W_{td} or W_{tt}, there is no guarantee that the singular vectors will remain the same. This result shows that, without applying any matrix normalization, performing LSI (while choosing the compressed representations as **only** the singular vectors) on the term-document matrix is equivalent to performing LSI on the word co-occurence matrix.

Ok but efficient for TF-IDF on W_{td} and then compute W_{tt} from this.

2. Studying the representation space

Without LSA, with the full tf-idf matrix

With the Euclidian distance between vectors

```
examples= ['the', 'dog', '3', 'good']
lookup_matrix = transform_tfidf(td_matrix)
show_examples = [vectorizer.tokenizer.word_to_token[word] for word in examples]
#print('the' in vectorizer.tokenizer.word_to_token)
```

```
#print('the' in vectorizer.tokenizer.word_to_token)
lab_util.show_similar_words(vectorizer.tokenizer, lookup_matrix, show_examples)
```

```
the 36
  . 0.331
  <unk> 0.366
  of 0.395
  and 0.403
  to 0.422
dog 925
  food 1.054
  pets 1.208
  pet 1.211
  foods 1.269
  dogs 1.314
3 289
  . 1.242
  8 1.252
  the 1.275
  to 1.276
  <unk> 1.282
good 47
  . 1.056
  a 1.102
  but 1.121
  , 1.152
  the 1.157
```

With cosine distance between vectors

```
def show_similar_words_cosine(tokenizer, reps, tokens):
    reps = reps / (np.sqrt((reps ** 2).sum(axis=1, keepdims=True)))
    #for i, (word, token) in enumerate(tokenizer.word_to_token.items()):
    for token in tokens:
        word = tokenizer.token_to_word[token]
        rep = reps[token, :]
        sims = np.sum(reps*rep, axis=1)
        nearest = np.argsort(sims)
        print(word, token)
        for j in nearest[-6:-1]:
            print(" ", tokenizer.token_to_word[j], "%.3f" % sims[j])

show_similar_words_cosine(vectorizer.tokenizer, lookup_matrix, show_examples)
```

```
the 36
  to 0.789
  and 0.799
  of 0.802
  <unk> 0.817
  . 0.834
dog 925
  dogs 0.343
  foods 0.365
```

```
    pet 0.394
    pets 0.396
    food 0.473
  3 289
    <unk> 0.359
    to 0.362
    the 0.363
    8 0.374
    . 0.379
  good 47
    the 0.421
    , 0.424
    but 0.440
    a 0.449
    . 0.472
```

Why are they the same ? FOr a vector u such that $||u|| = 1$,
$||u - v||^2 = ||u||^2 + ||v||^2 - 2 < u, v >$: $\min\limits_{v,||v||=1} ||u - v||^2 \iff \min\limits_{v,||v||=1} - < u, v > \iff$

Therefore, minimizing the Euclidian distance is equivalent to maximizing the cosine distance (after a Normalization step).

Back to desirata for distributional semantics, we will go over the different consttraints we wished to be satisfied for our word representation and check whether there are any evidences of check/fail:

- types: Our word representations should capture information about types
- constraints on predicate-argument relations:

Types

For nouns

```
examples= ['carrots', 'dog', 'surprise', 'heart']
show_examples = [vectorizer.tokenizer.word_to_token[word] for word in examples]
lab_util.show_similar_words(vectorizer.tokenizer, lookup_matrix, show_examples)
```

```
  carrots 943
    pure 1.033
    marketing 1.182
    peas 1.237
    labeled 1.323
    carrot 1.326
  dog 925
    food 1.054
    pets 1.208
    pet 1.211
    foods 1.269
```

```
    dogs 1.314
  surprise 961
    poured 1.585
    custard 1.618
    iams 1.707
    shelves 1.710
    140 1.719
  heart 963
    purina 1.056
    greta 1.057
    bone 1.264
    busy 1.281
    death 1.314
```

For verbs

```
examples= ['certified', 'means', 'write', 'tried']
show_examples = [vectorizer.tokenizer.word_to_token[word] for word in examples]
lab_util.show_similar_words(vectorizer.tokenizer, lookup_matrix, show_examples)
```

```
  certified 7
    dop 0.502
    marzano 0.876
    tomatoes 0.959
    p 1.216
    san 1.251
  means 9
    clear 1.449
    processed 1.521
    turns 1.563
    below 1.575
    heat 1.579
  write 22
    greta 1.106
    purina 1.151
    death 1.241
    bone 1.302
    busy 1.318
  tried 75
    i 1.256
    . 1.334
    and 1.345
    it 1.369
    the 1.402
```

We can see that our representation well captures information about types, this comes from the fact same similar types words should occur in the same situation.

Influence of the size of the LSA representation on representation space: Visualizing Representation Space with LSI (LSA on TFIDF)

One interesting thing to notice is that the Euclidian distance between words and their surroundings allow to know how good we surroung a word in High-Dimensionality. But one pattern that we start discovering is that **the closest words are actually far away**. This is due to the fact that, in high dimensions we are all alone: Curse of Dimensionality (Folks theorem). Therefore, when we increase the size of LSA, we will account for more features and allow for more separation and maybe a better clustering, but we will prevent our model from learning 'stratiications' in the representation space. Reducing the size of LSA will create more 'compact' representations, but maybe we will lose some information by projecting into a subspace and letting go some information brought by marginal singular vectors (maybe their own contribution is low, but this technique prevents from considering multivariate effect). Therefore, let us choose this trade-off by visualizing the scaled variance of every eigenvalue, being a proxy for the contribution of the corresponding eigenvector.

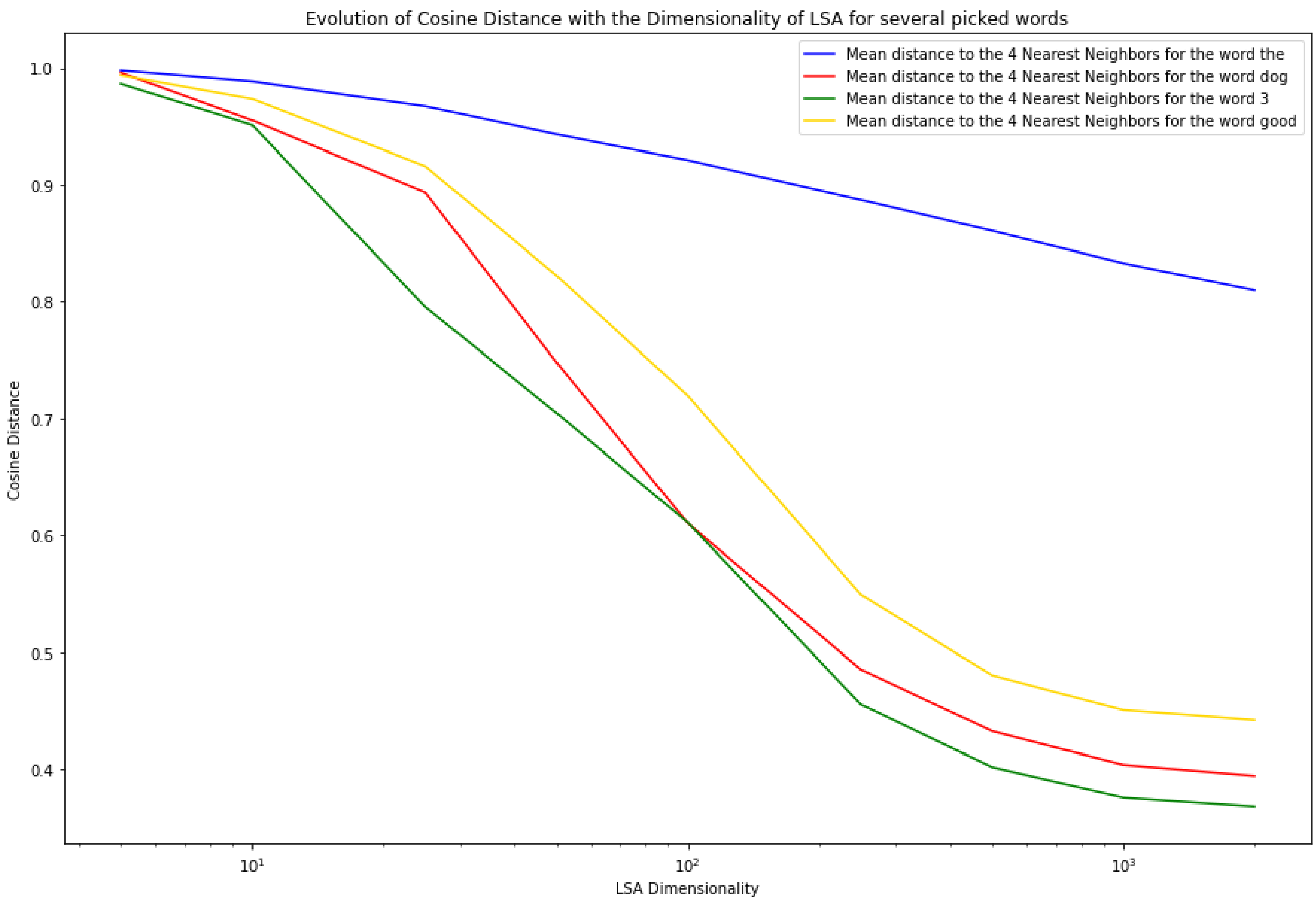
Neighborhood in High Dimensions and effect of Dimensionality

For this experiment, we are going to choose the 4 words the, dog, 3, and good and check how the mean distance to their 5 closest neighbors evolve with the dimensionality of the LSA.

```
%%time
examples= ['the', 'dog', '3', 'good']
show_examples = [vectorizer.tokenizer.word_to_token[word] for word in examples]
dims = [5, 10, 25, 50, 100, 250, 500, 1000, 2000]
list_distances_dims = np.zeros((4, 9))
for i, dim in enumerate(dims):
  reps = learn_reps_lsa(td_matrix_tfidf, rep_size=dim)
  reps = reps / (np.sqrt((reps ** 2).sum(axis=1, keepdims=True)))
  for j, token in enumerate(show_examples):
      word = vectorizer.tokenizer.token_to_word[token]
      rep = reps[token, :]
      sims = np.sum(reps*rep, axis=1)
      nearest = np.argsort(sims)
      mean_distance = np.mean([sims[j] for j in nearest[-6:-1]])
      list_distances_dims[j, i] = mean_distance
  print('Done with dimension', dim)
```

```
Done with dimension 5
Done with dimension 10
Done with dimension 25
Done with dimension 50
Done with dimension 100
Done with dimension 250
Done with dimension 500
Done with dimension 1000
Done with dimension 2000
CPU times: user 1min 14s, sys: 35.9 s, total: 1min 50s
Wall time: 28.2 s
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, figsize = (15, 10))
ax.plot(dims, list_distances_dims[0], color='blue', label='Mean distance to the 4 Nearest Nei
ax.plot(dims, list_distances_dims[1], color='red', label='Mean distance to the 4 Nearest Neig
ax.plot(dims, list_distances_dims[2], color='green', label='Mean distance to the 4 Nearest Ne
ax.plot(dims, list_distances_dims[3], color='gold', label='Mean distance to the 4 Nearest Nei
ax.legend()
ax.set_xlabel('LSA Dimensionality')
ax.set_ylabel('Cosine Distance')
plt.title('Evolution of Cosine Distance with the Dimensionality of LSA for several picked wor
ax.set_xscale('log')
plt.show(fig)
```



Therefore, we can clearly see that dimensionality affects distance to nearest neighbours: but does it affect representation, ie mutual distances, ie the neighborhood of every point. The question would be: is varying the dimensionality of LSA having an influence on which words are closest to each others ?

```
%%time
examples= ['the', 'dog', '3', 'good']
show_examples = [vectorizer.tokenizer.word_to_token[word] for word in examples]
dims = [5, 10, 25, 50, 100, 250, 500, 1000, 2000]
list_distances_dims = np.zeros((4, 9))
closest_point_dims = []
distances = []
for i, dim in enumerate(dims):
  reps = learn_reps_lsa(td_matrix_tfidf, rep_size=dim)
  reps = reps / (np.sqrt((reps ** 2).sum(axis=1, keepdims=True)))
  for j, token in enumerate(show_examples):
      word = vectorizer.tokenizer.token_to_word[token]
      rep = reps[token, :]
      sims = np.sum(reps*rep, axis=1)
      nearest = np.argsort(sims)
      mean_distance = np.mean([sims[j] for j in nearest[-6:-1]])
      closest_point = vectorizer.tokenizer.token_to_word[nearest[-2]]
      closest_point_dims.append(closest_point)
      distances.append(sims[nearest[-2]])
      list_distances_dims[j, i] = mean_distance
  print('Done with dimension', dim)
```

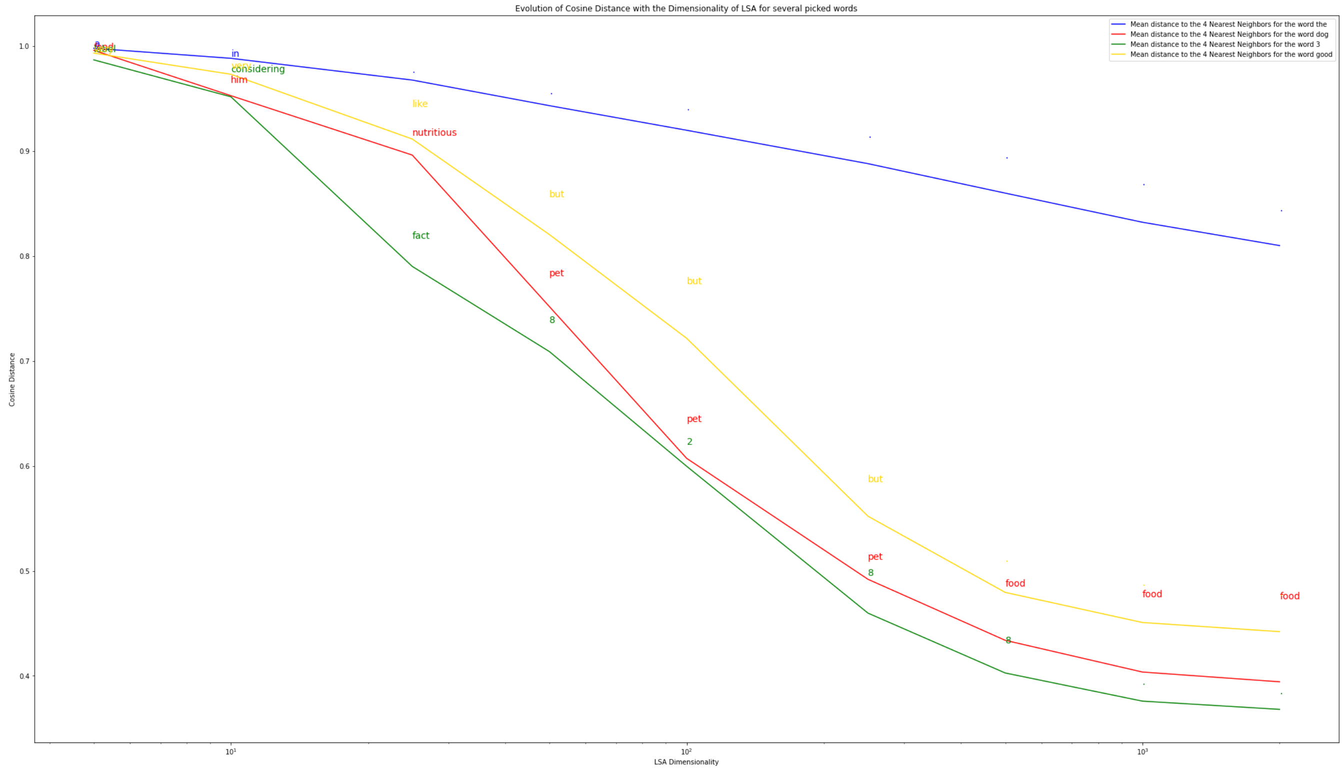
```
Done with dimension 5
Done with dimension 10
Done with dimension 25
Done with dimension 50
Done with dimension 100
Done with dimension 250
Done with dimension 500
Done with dimension 1000
Done with dimension 2000
CPU times: user 1min 15s, sys: 36 s, total: 1min 51s
Wall time: 28.3 s
```

```
array_words = np.array([closest_point_dims[i:i+4] for i in range(0, len(closest_point_dims),
print(array_words)
distances_y = np.array([distances[i:i+4] for i in range(0, len(distances), 4)]).T
print(distances_y)
```

```
[['a' 'in' '.' '.' '.' '.' '.' '.' '.']
 ['food' 'him' 'nutritious' 'pet' 'pet' 'pet' 'food' 'food' 'food']
 ['label' 'considering' 'fact' '8' '2' '8' '8' '.' '.']
 ['very' 'very' 'like' 'but' 'but' 'but' '.' '.' '.']]
[[0.9997048  0.99015261 0.97484618 0.95410351 0.93882325 0.91284098
  0.89337425 0.86797435 0.84288926]
 [0.99636231 0.96547735 0.91491745 0.78068292 0.64175332 0.51082409
  0.48507241 0.47481732 0.47295266]
```

```
      [0.994494   0.97532131 0.8165439  0.73630095 0.62050736 0.4952414
       0.43132583 0.39163897 0.38261811]
      [0.99459014 0.97836178 0.94216799 0.8564467  0.77303972 0.58449109
       0.50895674 0.48613677 0.47655606]]
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, figsize = (35, 20))
colors = ['blue', 'red', 'green', 'gold']
ax.plot(dims, list_distances_dims[0], color='blue', label='Mean distance to the 4 Nearest Nei
ax.plot(dims, list_distances_dims[1], color='red', label='Mean distance to the 4 Nearest Neig
ax.plot(dims, list_distances_dims[2], color='green', label='Mean distance to the 4 Nearest Ne
ax.plot(dims, list_distances_dims[3], color='gold', label='Mean distance to the 4 Nearest Nei
for k in range(4):
  for i, dim in enumerate(dims):
    word = array_words[k, i]
    distance = distances_y[k, i]
    ax.annotate(s=str(word), xy=(dim, distance), color=colors[k], size=14)
ax.legend()
ax.set_xlabel('LSA Dimensionality')
ax.set_ylabel('Cosine Distance')
plt.title('Evolution of Cosine Distance with the Dimensionality of LSA for several picked wor
ax.set_xscale('log')
plt.show(fig)
```



From this plot, several interesting things appear:

- As expected, when truncating into a very low dimensional space, all the words are very close to another and the closest neighborhood is at cosine distance 1.
- When increasing the dimensionality of the LSA representation, the closest neighbours seem do make more sense, with syntactic similarities between words and even meaningful nearest neighbors
- When the size of the LSA is too big, nearest neighbours do not make much sense anymore, and they are all closely related to stop words, which occur everywhere
- This confirms our intuition that there is a serious trade-off in finding the optimal dimension for the LSA

Finding the optimal dimension for the LSA : heuristic with the cumulated variance

Since the LSA is done on the tf-idf matrix, we are going to apply our heuristic on this matrix. We know that the singular values can be expressed as the square roots of the eigenvalues of the empirical covariance matrix.

Implementation detail: we take the absolute value because we know that XX^T is SDP (where X is the tf-idf matrix), so the eigenvalues must be positive

```
singular_values_sorted = sorted(np.sqrt(np.abs(np.linalg.eigvals(td_matrix_tfidf@td_matrix_tf
normalized_singular_values = singular_values_sorted/np.sum(singular_values_sorted)
```

```
cumulated_sum = np.cumsum(normalized_singular_values)
print('For retaining 80% of the variance, we would need ' + str(np.sum(cumulated_sum < 0.8))
print('For retaining 90% of the variance, we would need ' + str(np.sum(cumulated_sum < 0.9))
```

```
For retaining 80% of the variance, we would need 1028 dimension for LSA
For retaining 90% of the variance, we would need 1355 dimension for LSA
```

Based on cumulated varince ratio, it seems like we would want to retain roughly 1000 dimensions, which still remains huge. Let us use another criteria for determinig the number of dimensions: the marginal contribution of every dimension

```
marginal_increases = (normalized_singular_values[:-1] - normalized_singular_values[1:])
print('We would need to retain ' + str(np.sum(marginal_increases > np.mean(marginal_increases
```

```
We would need to retain 94 dimension for LSA
```

That seems way better !

Effect on downstream, classification task

Performances of the classification task with LSI

As we have seen before, there is definitely a trade off in dimensionality reduction: we need to find the 'sweet-spot'. Before, we did that in an Unsupervised Way using variance retained. Now that we have a downstream classification task, we could fine-tune our representations based on this classification task. We are going to do that on the validation set, and find which is the effect of representation on classification.We will be working with the combo_featurizer in order to understand the contribution of both vectorized words and LSI representations. Why working on the validation set ? Because here we interpret the size of LSI as a hyperparameter, and the validation set is kept for Hyperparameter tuning.

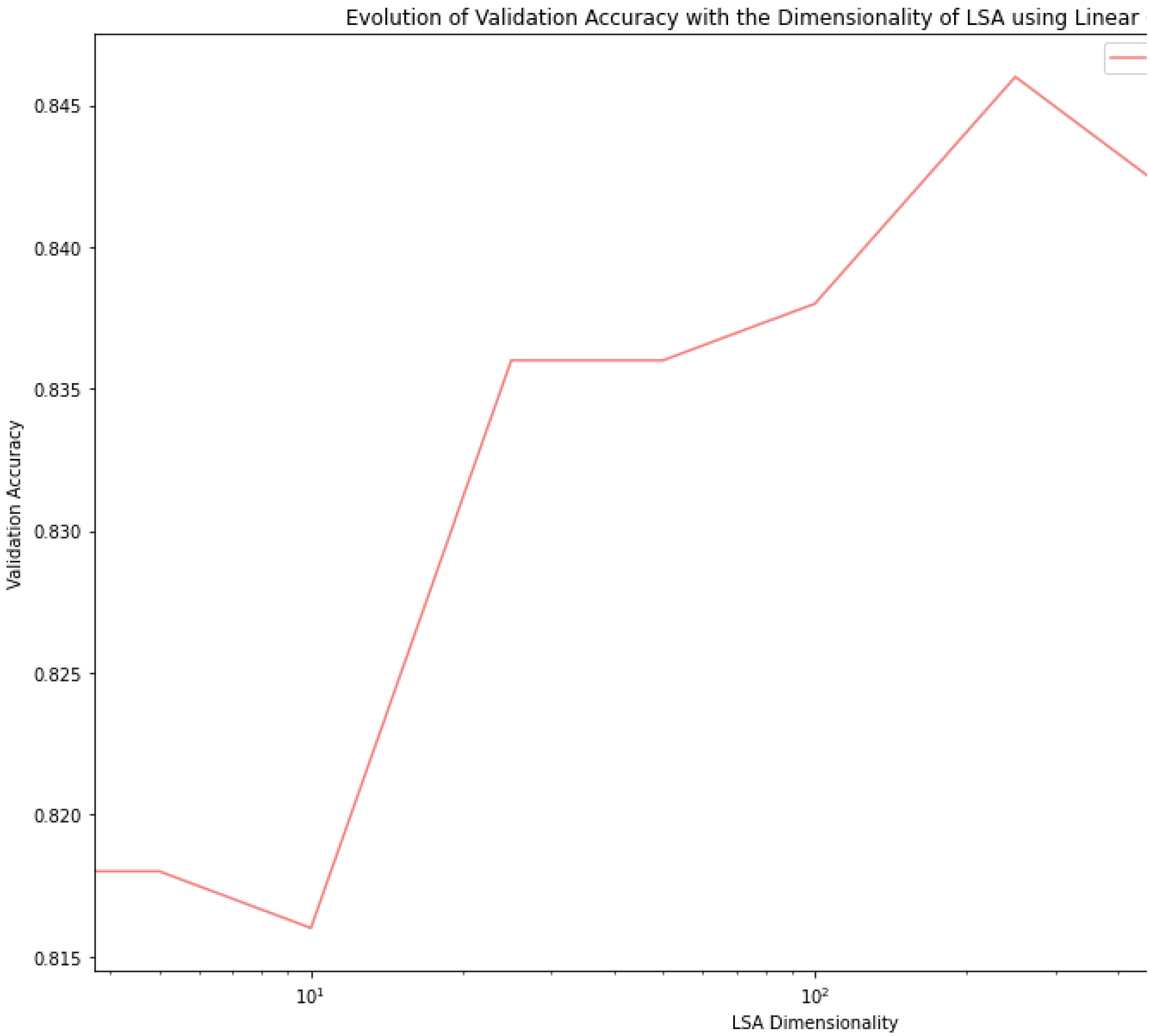
```
%%time
import sklearn.linear_model

td_matrix_tfidf = transform_tfidf(td_matrix)  # look-up table for the training embeddings

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

def lsa_featurizer(xs, dims=1000):
    # 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).
    features = learn_reps_lsa(td_matrix_tfidf, dims)
    # now, inside features [in rows] we have the embeddings for every word
    resulting_embeddings = []
    for i, review in enumerate(xs):
      review_embeddings = []
      for j, word in enumerate(review):
        if word > 0: # word, with token j is present in review i
          review_embeddings.append(features[j])
      resulting_embeddings.append(np.mean(review_embeddings, axis=0))
    feats = np.array(resulting_embeddings)
    # normalize
    return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

# We've implemented the remainder of the training and evaluation pipeline,
# so you likely won't need to modify the following four functions.
def combo_featurizer(xs, dimension):
    return np.concatenate((word_featurizer(xs, 0), lsa_featurizer(xs, dimension)), axis=1)

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

def eval_model(model, featurizer, xs, ys, dimension):
    xs_featurized = featurizer(xs, dimension)
    pred_ys = model.predict(xs_featurized)
    return np.mean(pred_ys == ys)

def training_experiment_2(name, featurizer, n_train, dimension):
    print(f"{name} features, {n_train} examples")
    # print(train_reviews[0])
    train_xs = vectorizer.transform(train_reviews[:n_train])
```

```
    train_ys = train_labels[:n_train]
    val_xs = vectorizer.transform(val_reviews)
    val_ys = val_labels
    model = train_model(featurizer, train_xs, train_ys, dimension)
    acc = eval_model(model, featurizer, val_xs, val_ys, dimension)
    print(acc, '\n')
    return acc

dimensions =  [0, 5, 10, 25, 50, 100, 250, 500, 1000, 2000]
validation_scores = [0.818]
training_experiment("word", word_featurizer, n_train, 0)
for dim in dimensions[1:]:
  vals = training_experiment("combo", combo_featurizer, n_train, dim)
  validation_scores.append(vals)
```

```
word features, 3000 examples
0.818

combo features, 3000 examples
0.818

combo features, 3000 examples
0.816

combo features, 3000 examples
0.836

combo features, 3000 examples
0.836

combo features, 3000 examples
0.838

combo features, 3000 examples
0.846

combo features, 3000 examples
0.842

combo features, 3000 examples
0.844

combo features, 3000 examples
0.846

CPU times: user 3min 15s, sys: 1min 19s, total: 4min 35s
Wall time: 1min 34s
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, figsize = (15, 10))
ax.plot(dimensions, validation_scores, label='Validation score of the Combo Embedding', color
ax.legend()
ax.set_xlabel('LSA Dimensionality')
ax.set_ylabel('Validation Accuracy')
plt.title('Evolution of Validation Accuracy with the Dimensionality of LSA using Linear Class
```

```
plt.title('Evolution of Validation Accuracy with the Dimensionality of LSA using Linear Class
ax.set_xscale('log')
plt.show(fig)
```



Therefore, two conclusions could be made from this visualization:

- The learned representations could help the classification task: this was one of our desiderata (when creating the embeddings in the context of a downstream classification task)
- This is an efficient way to select the optimal dimensionality for LSI, when used in the context if a downstream task

Relationship between number of labeled examples and effect of word embeddings

Result

Intuition: what happens when the number of labeled data diminishes ? The compression becomes **less** efficient. Indeed, let us have a toy example: 10000 examples and LSI with 1000 dimensions: we have a 10 $\times$ compression and the information will be efficiently encoded efficiently. However, if we have 1000 examples and LSI with 1000 dimensions: data will be very sparse. Therefore, intuitively, allowing for a large vocabulary size (brought by a big number of reviews) allow our unsupervised learning model to learn some patterns in rich structures. Therefore, I think that the effect of representations will be less effective when having less labelled data points.

```
%%time
import sklearn.linear_model

td_matrix_tfidf = transform_tfidf(td_matrix)  # look-up table for the training embeddings

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

def lsa_featurizer(xs, dims=1000):
    # 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).
    features = learn_reps_lsa(td_matrix_tfidf, dims)
    # now, inside features [in rows] we have the embeddings for every word
    resulting_embeddings = []
    for i, review in enumerate(xs):
      review_embeddings = []
      for j, word in enumerate(review):
        if word > 0: # word, with token j is present in review i
          review_embeddings.append(features[j])
      resulting_embeddings.append(np.mean(review_embeddings, axis=0))
    feats = np.array(resulting_embeddings)
    # normalize
    return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

# We've implemented the remainder of the training and evaluation pipeline,
# so you likely won't need to modify the following four functions.
def combo_featurizer(xs, dimension):
    return np.concatenate((word_featurizer(xs, 0), lsa_featurizer(xs, dimension)), axis=1)

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

```
        return model

    def eval_model(model, featurizer, xs, ys, dimension):
        xs_featurized = featurizer(xs, dimension)
        pred_ys = model.predict(xs_featurized)
        return np.mean(pred_ys == ys)

    def training_experiment(name, featurizer, n_train, dimension):
        print(f"{name} features, {n_train} examples")
        # print(train_reviews[0])
        train_xs = vectorizer.transform(train_reviews[:n_train])
        train_ys = train_labels[:n_train]
        val_xs = vectorizer.transform(val_reviews)
        val_ys = val_labels
        model = train_model(featurizer, train_xs, train_ys, dimension)
        acc = eval_model(model, featurizer, val_xs, val_ys, dimension)
        print(acc, '\n')
        return acc

    dimensions =  [0, 5, 10, 25, 50, 100, 250, 500, 1000, 2000]
    #validation_scores = [0.818]
    #training_experiment("word", word_featurizer, n_train, 0)
    n_trains = [500, 1000, 2000, 3000]
    for n_train in n_trains:
      for dim in dimensions:
        if dim < n_train:
          vals = training_experiment("combo", combo_featurizer, n_train, dim)
          validation_scores.append(vals)
```

```
combo features, 500 examples
0.754

combo features, 500 examples
0.754

combo features, 500 examples
0.754

combo features, 500 examples
0.756

combo features, 500 examples
0.762

combo features, 500 examples
0.77

combo features, 500 examples
0.768

combo features, 1000 examples
0.776

combo features, 1000 examples
```

```
0.776

combo features, 1000 examples
0.77

combo features, 1000 examples
0.782

combo features, 1000 examples
0.784

combo features, 1000 examples
0.788

combo features, 1000 examples
0.786

combo features, 1000 examples
0.786

combo features, 2000 examples
0.798

combo features, 2000 examples
0.798

combo features, 2000 examples
0.8

combo features, 2000 examples
0.806

combo features, 2000 examples
0.81
```

Why is it important: Unsupervised Learning

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 matrix factorization objective. This requires a little more work; we've provided scaffolding for a PyTorch model implementation below. If you don't have much PyTorch experience, there are some tutorials [here](#) which may be useful. You're also welcome to implement these experiments in any other framework of your choosing.

```
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
```

```
device = torch.device('cuda')


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.
    # here, we implement the CBOW version

    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.l1 = nn.Linear(vocab_size, embed_dim)
        self.l2 = nn.Linear(embed_dim, vocab_size)

    def forward(self, context):  # we need to one hot encode all of the words inside it,
        # 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
        #context.to('cpu')
        resulting_tensor = None
        for individual_context in context:
          one_hot = torch.nn.functional.one_hot(individual_context[individual_context >= 0],
          one_hot_ind = torch.mean(one_hot, dim=0)
          if resulting_tensor is None:
            resulting_tensor = one_hot_ind
          else:
            resulting_tensor = torch.vstack([resulting_tensor, one_hot_ind])
        output = self.l1(resulting_tensor)
        output = self.l2(output)
        # Your code here!
        return output


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)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

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

    # What loss function should we use for Word2Vec?
    loss_fn = nn.CrossEntropyLoss()  # Your code here!

    losses = []  # Potentially useful for debugging (loss should go down!)
    for epoch in tqdm(range(n_epochs)):
        epoch_loss = 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.
            optimizer.zero_grad()
            # Here, perform a forward pass to compute predictions for the model.
            preds = model(context.to(device))  # Your code here!
            # Now finish the backward pass and gradient update.
            # Remember, you need to compute the loss, zero the gradients
            # of the model parameters, perform the backward pass, and
            # update the model parameters.
            loss = loss_fn(preds, label.to(device))  # Your code here!
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
        losses.append(epoch_loss)

    # Hint: you want to return a `vocab_size x embedding_size` numpy array
    embedding_matrix = model.l1.weight

    return embedding_matrix


# Use the function you just wrote to learn Word2Vec embeddings:
reps_word2vec = learn_reps_word2vec(train_reviews, 2, 500, 10, 100).to('cpu').detach().numpy(
```

```
100%|██████████| 10/10 [09:02<00:00, 54.24s/it]
```

After training the embeddings, we can try to visualize the embedding space to see if it makes sense. First, we can take any word in the space and check its closest neighbors.

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

```
good 47
  great 0.971
  decent 1.031
  bad 1.131
  disgusting 1.181
  fantastic 1.182
bad 201
  good 1.131
  awful 1.168
  bitter 1.199
```

```
    overpowering 1.274
    overwhelming 1.284
  cookie 504
    g 1.293
    covered 1.363
    berry 1.376
    wheat 1.399
    nana's 1.447
  jelly 351
    sized 1.226
    bears 1.243
    pork 1.293
    candies 1.309
    san 1.344
  dog 925
    cat 0.852
    baby 1.011
    cats 1.202
    canned 1.240
    husband 1.304
  the 36
    mrs 1.080
    my 1.216
    our 1.221
    their 1.238
    amazon's 1.252
  4 292
    2 0.798
    5 0.924
    10 0.985
    75 1.108
    3 1.133
```

We can also cluster the embedding space. Clustering in 4 or more dimensions is hard to visualize, and even clustering in 2 or 3 can be difficult because there are so many words in the vocabulary. One thing we can try to do is assign cluster labels and qualitiatively look for an underlying 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}")
```

```
careful: 0
having: 0
craving: 0
exact: 0
```

```
listed: 0
making: 0
happy: 0
giving: 0
thrown: 0
doing: 0
available: 0
must: 1
them: 1
once: 1
usually: 1
how: 1
puck: 1
everywhere: 1
until: 1
perfectly: 1
extremely: 1
probably: 1
dessert: 2
coffees: 2
gravy: 2
soy: 2
pouch: 2
baked: 2
carrot: 2
mostly: 2
seasoning: 2
jalapeno: 2
gourmet: 2
cool: 3
wellness: 3
late: 3
gift: 3
roast: 3
market: 3
second: 3
stomach: 3
favorite: 3
tassimo: 3
usual: 3
spot: 4
helped: 4
didn't: 4
through: 4
sent: 5
learned: 5
experienced: 5
started: 5
watered: 6
broken: 6
disgusting: 6
acidic: 6
spicy: 6
moist: 6
somewhat: 6
```

Finally, we can use the trained word embeddings to construct vector representations of full reviews. One common approach is to simply average all the word embeddings in the review to create an overall embedding. Implement the transform function in Word2VecFeaturizer to do this.

```
def w2v_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 average Word2Vec embedding of each review (hint: this will be very
    # similar to `lsa_featurizer` from above, just using Word2Vec embeddings
    # instead of LSA).
    features = reps_word2vec
    # now, inside features [in rows] we have the embeddings for every word
    resulting_embeddings = []
    for i, review in enumerate(xs):
      review_embeddings = []
      for j, word in enumerate(review):
        if word > 0: # word, with token j is present in review i
          review_embeddings.append(features[j])
      resulting_embeddings.append(np.mean(review_embeddings, axis=0))
    feats = np.array(resulting_embeddings)
    # normalize
    return feats / np.sqrt((feats ** 2).sum(axis=1, keepdims=True))

training_experiment_1("word2vec", w2v_featurizer, 3000)
print()
```

```
word2vec features, 3000 examples
0.766
```

Part 2: Lab writeup

Part 2 of your lab report should discuss any implementation details that were important to filling out the code above, as well as your answers to the questions in Part 2 of the Homework 1 handout. Below, you can set up and perform experiments that answer these questions (include figures, plots, and tables in your write-up as you see fit).

Experiments for Part 2

Language Modeling

Intuition

From the different experiments, two patterns seem to emerge from the Word2Vec representations:

- they seem **very relevant** in terms of association. In the embedding spaces, words seem to be closer to other ones that have same meaning (or at least the same function in the sentence).
- Moreover, the KMeans clusters created on the algorithm seem to correspond to the following classes:
  - Cluster 1: the very common words
  - Cluster 2: the very common nouns
  - Cluster 3: verbs used in a present context
  - Cluster 4: complements to the verbs (terms of action)
  - Cluster 5: verb adjectives
  - Cluster 6: nouns related to food
  - Cluster 7: ?
  - Cluster 8: verbs used in a past context
  - Cluster 9: adjectives used in a context of quantity
  - Cluster 10: Miscellaneous
- I think that these representations are very intuitive and seem to encode **word similarity**, in the sense that words that could be used in an exchangeable way (ie without changing the **syntactic correctness of the sentence**) seem to be inside the same category.

Verifying the linear structure: Additive Compositionality

In the initial paper released by Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*, we saw that non-linearities were **not used** because it allows the represntations to present a linear pattern. Let us verify if this also happens in this representation.

```
def substraction(word1, word2):
  """This function takes two words present in the corpus and computes word1 - word2 in the wc
  token1 = vectorizer.tokenizer.word_to_token[word1]
  token2 = vectorizer.tokenizer.word_to_token[word2]
  rep_substract = reps_word2vec[token1] - reps_word2vec[token2]
  sims = ((reps_word2vec - rep_substract) ** 2).sum(axis=1)
  nearest = np.argsort(sims)
  return vectorizer.tokenizer.token_to_word[nearest[1]]
```

```
substraction('hard', 'easy')
```

```
'<unk>'
```

```
substraction('fat', 'eat')
```

```
'protein'
```

Analogical Reasoning

```
lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, [vectorizer.tokenizer.word_t
```

```
omega 514
  potassium 1.152
  iron 1.167
  2012 1.214
  cholesterol 1.245
  sodium 1.292
```

```
lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, [vectorizer.tokenizer.word_t
```

```
times 290
  months 1.146
  years 1.166
  weeks 1.174
  days 1.214
  hours 1.228
```

```
lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, [vectorizer.tokenizer.word_t
```

```
save 27
  waste 1.318
  spend 1.355
  avoid 1.376
  pay 1.386
  wait 1.398
```

Downstream Classification as a benchmarking method

Analysis of Word2Vec Representation

Here, since we only use the word2vec embeddings for the representation, we will compare the performances of using only word2vec embeddings with the performances of using only lsa featurizer.

We can see that the performances of using word2vec are slightly better than when using lsa_featurizer. However, the advantages of using word2vec are:

- More flexibility in the representation
- Tractable computations in a NN
- Results more interpretable

Last, we still haven't taken into account a fundamental parameter in the word2vec implementation: the **embedding size** (which was equivalent, in the LSA to the dimension of the truncated SVD).

Let us Cross-Validate on a few different embedding dimension, and check which influence does this parameter have on the downstream task.

Influence of the Embedding dimension

Our hope is that after fine-tuning the embedding dimension, we have the opportunity to improve a lot our classification performances, and get significantly better results woth the Word2Vec embeddings than with the LSA embeddings. A priori, here is my thought: the embedding dimension becomes the number of features for the Linear classifier. Since we have onlty 3000 training examples, selecting the embedding dimension as being greater than 3000 creates an over-specified problem and raises issues with computational stability of the estimates: we don't want tot do that. For smaller embedding dimensions, when we increase the number of dimensions, it becomes harder for our classifier to learn useful patterns in our data (more fetaures, same amout on training data): this is where the tradeoff happens: there is a need to select carefully the number of dimensions.

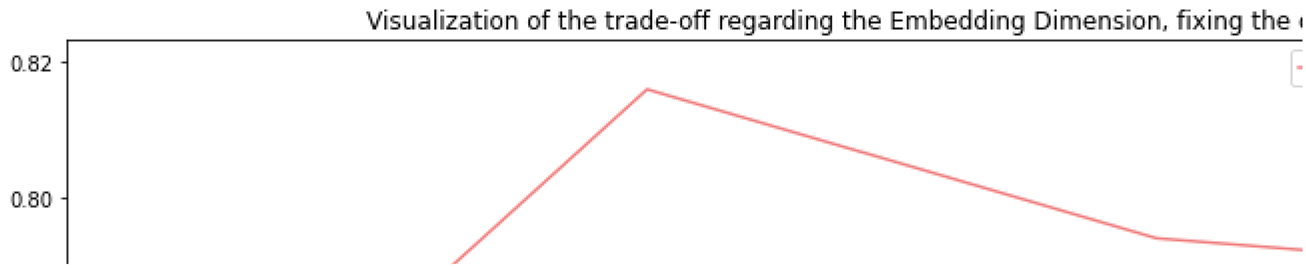
```
for dim in [10, 50, 100, 250, 500, 1000]:
  reps_word2vec = learn_reps_word2vec(train_reviews, 2, dim, 10, 100).to('cpu').detach().nump
  training_experiment_1("word2vec", w2v_featurizer, 3000)
```

```
100%|██████████| 10/10 [13:57<00:00, 83.78s/it]
word2vec features, 3000 examples
0.67

100%|██████████| 10/10 [13:58<00:00, 83.89s/it]
word2vec features, 3000 examples
0.746

100%|██████████| 10/10 [13:58<00:00, 83.89s/it]
word2vec features, 3000 examples
0.75

100%|██████████| 10/10 [14:00<00:00, 84.03s/it]
word2vec features, 3000 examples
0.776

100%|██████████| 10/10 [14:02<00:00, 84.21s/it]
word2vec features, 3000 examples
0.782

100%|██████████| 10/10 [14:08<00:00, 84.89s/it]
word2vec features, 3000 examples
0.816
```

```
for dim in [2000, 3000]:
  reps_word2vec = learn_reps_word2vec(train_reviews, 2, dim, 10, 100).to('cpu').detach().nump
  training_experiment_1("word2vec", w2v_featurizer, 3000)
```

```
100%|██████████| 10/10 [09:44<00:00, 58.46s/it]
word2vec features, 3000 examples
0.794

100%|██████████| 10/10 [10:16<00:00, 61.64s/it]
word2vec features, 3000 examples
0.792
```

```
import matplotlib.pyplot as plt

embedding_dimension =  [10, 50, 100, 250, 500, 1000, 2000, 3000]
scores = [0.67, 0.746, 0.75, 0.776, 0.782, 0.816, 0.794, 0.788]
fig, ax = plt.subplots(1, figsize =(15, 10))
ax.plot(embedding_dimension, scores, label='Test Accuracy of the Linear Classifier', color='
ax.set_xlabel('Embedding dimension, context size fixed')
ax.set_ylabel('Test Accuracy')
plt.legend()
ax.set_title('Visualization of the trade-off regarding the Embedding Dimension, fixing the co
plt.show()
```



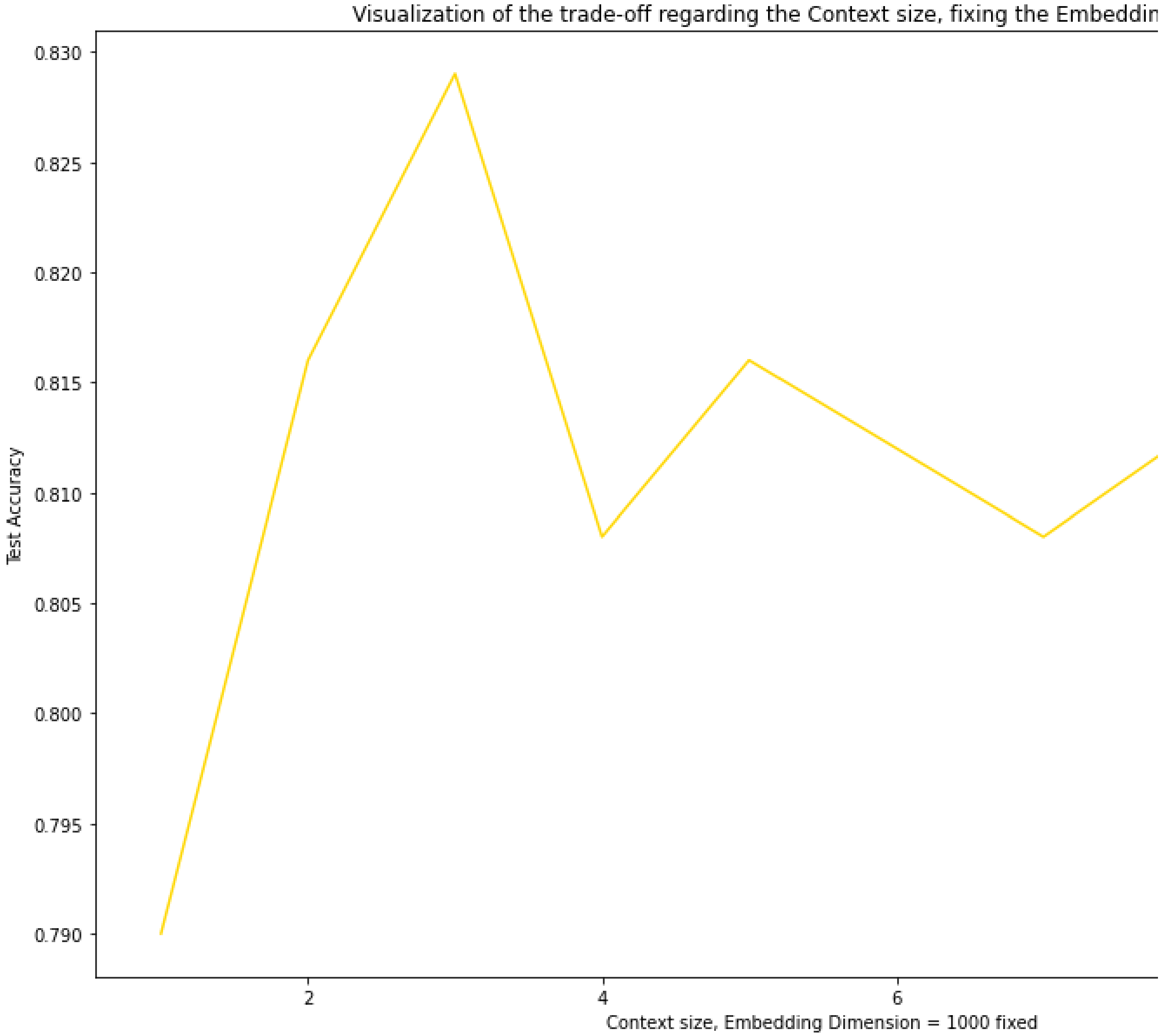
We can see that the results confirm or intuition regaridng the Embedding dimension. We even get results that are significantly higher with Word2Vec Embeddings than with LSI Embeddings. Another hyperparameter that might affect performances of Word2Vec is the **context size**. Let us explore how this parameter might affect performances, and later on we will see its influence on the *representation*. So let's perform the following experiment: fixing the embedding dimension to 1000, we will vary the context size and check how this affects performances.

```
for context_size in [1, 3, 4, 5, 7, 10]:
  reps_word2vec = learn_reps_word2vec(train_reviews, context_size, 1000, 10, 100).to('cpu').c
  training_experiment_1("word2vec", w2v_featurizer, 3000)
```

```
100%|██████████| 10/10 [09:14<00:00, 55.49s/it]
word2vec features, 3000 examples
0.79

100%|██████████| 10/10 [09:18<00:00, 55.85s/it]
word2vec features, 3000 examples
0.808

100%|██████████| 10/10 [09:19<00:00, 55.96s/it]
word2vec features, 3000 examples
0.808

100%|██████████| 10/10 [09:20<00:00, 56.02s/it]
word2vec features, 3000 examples
0.816

100%|██████████| 10/10 [09:20<00:00, 56.08s/it]
word2vec features, 3000 examples
0.808

100%|██████████| 10/10 [09:22<00:00, 56.29s/it]
word2vec features, 3000 examples
0.822
```

```
context_size =  [1, 2, 3, 4, 5, 7, 10]
scores = [0.79, 0.816, 0.829, 0.808, 0.816, 0.808, 0.822]
fig, ax = plt.subplots(1, figsize =(15, 10))
ax.plot(context_size, scores, label='Test Accuracy of the Linear Classifier', color='gold')
ax.set_xlabel('Context size, Embedding Dimension = 1000 fixed')
```

```
ax.set_ylabel('Test Accuracy')
plt.legend()
ax.set_title('Visualization of the trade-off regarding the Context size, fixing the Embedding
plt.show()
```



Influence of the context size on the learned representations

```
for context_size in [1, 3, 4, 5, 7, 10]:
  reps_word2vec = learn_reps_word2vec(train_reviews, context_size, 1000, 10, 100).to('cpu').d
  training_experiment_1("word2vec", w2v_featurizer, 3000)
  lab_util.show_similar_words(vectorizer.tokenizer, reps_word2vec, show_tokens)
```

```
2 1.065
6 1.226
10 1.256
3 1.269
three 1.354
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
 10%|█         | 1/10 [01:28<13:13, 88.21s/it]
 20%|██        | 2/10 [02:56<11:45, 88.22s/it]
 30%|███       | 3/10 [04:24<10:17, 88.17s/it]
 40%|████      | 4/10 [05:52<08:48, 88.12s/it]
 50%|█████     | 5/10 [07:20<07:20, 88.04s/it]
 60%|██████    | 6/10 [08:48<05:51, 87.93s/it]
 70%|███████   | 7/10 [10:15<04:23, 87.94s/it]
 80%|████████  | 8/10 [11:43<02:55, 87.92s/it]
 90%|█████████ | 9/10 [13:11<01:27, 87.94s/it]
100%|██████████| 10/10 [14:39<00:00, 87.99s/it]
word2vec features, 3000 examples
0.822

good 47
  great 1.167
  fine 1.275
  decent 1.300
  bad 1.311
  happy 1.320
bad 201
  good 1.311
  horrible 1.411
  strong 1.425
  decent 1.433
  acidic 1.434
cookie 504
  box 1.292
  chip 1.334
  berry 1.366
  warning 1.439
  closer 1.440
jelly 351
  cardboard 1.301
  basil 1.364
  home 1.475
  kept 1.501
  cookies 1.503
dog 925
  cat 1.011
  dogs 1.077
  cats 1.107
  breed 1.152
  baby 1.196
the 36
  commercial 1.198
  alot 1.245
  a 1.273
  dead 1.285
  their 1.298
4 292
  2 1.264
  6 1.303
```