In [1]:
```
version = "v1.12.052622.1"
```

SIADS 543 Assignment 3: Text representations, topic modeling and word embeddings

In this week's assignment you'll gain experience applying topic modeling and other latent variable estimation methods. We'll focus on textual data, continuing to work with vectorizers and related text representations like embeddings.

All questions in this assignment are auto-graded. Some parts ask you a short question or two about on the results: these are meant to encourage you to reflect on the outcomes, but do not need to be included as part of your graded submission.

In [2]:
```
# First import some necessary libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline

# Suppress all warnings
import warnings

warnings.filterwarnings("ignore")

np.set_printoptions(precision=3)
```

Here are some useful utility functions to use with this assignment.

In [3]:
```
import pickle
from sklearn.feature_extraction.text import TfidfVectorizer, CountVector

# display_topics:  example showing how to take the model components gene
# and use them to dump the top words by weight for each topic.
def display_topics(model, feature_names, num_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print("Topic %d:" % (topic_idx))
        print(
            " ".join(
                [feature_names[i] for i in topic.argsort()[: -num_top_wo
            )
        )


# load_newsgroup_documents: prepare training and test data from the 20ne
def load_newsgroup_documents():
    # The Coursera environment must be self-contained and so APIs that d
    # aren't allowed. So we use pickle files that can be stored locally 
    # API calls.
    # dataset_train   = fetch_20newsgroups(subset = 'train', shuffle=Tru
    # dataset_test    = fetch_20newsgroups(subset = 'test', shuffle=True

    pickle_train_data = open("./assets/20newsgroups_train_data.pickle", 
    pickle_train_labels = open("./assets/20newsgroups_train_labels.pickl
    documents_train = pickle.load(pickle_train_data)
    labels_train = pickle.load(pickle_train_labels)
    pickle_train_data.close()
    pickle_train_labels.close()

    return documents_train, labels_train


# load the dataset for future use....
documents_train, labels_train = load_newsgroup_documents()
```

Question 1 (20 points) The choice of text processing can impact final classification performance.

There are many different parameter settings for Vectorizer objects in scikit-learn. Small changes in these settings can result in very different text representations and significant changes in final classifier accuracy. For this question you'll train a commonly-used type of text classifier, Multinomial Naive Bayes, using three different input representations for text, to see the effect of different parameter choices on classifier training set accuracy.

Follow these steps:

1. Create a TfidfVectorizer object (let's call it A) with the following settings:

   ```
   max_features = 10000, # only top 10k by freq

   lowercase = False, # keep capitalization

   ngram_range = (1,2), # include 2-word phrases

   min_df=10,  # note: absolute count of documents

   max_df=0.95,   # note: % of docs in collection

   stop_words='english'
   ```

2. Create a CountVectorizer object (let's call it B) with the same settings:

   ```
   max_features = 10000, # only top 10k by freq

   lowercase = False, # keep capitalization

   ngram_range = (1,2), # include 2-word phrases

   min_df=10,  # note: absolute count of doc

   max_df=0.95,   # note: % of docs

   stop_words='english'
   ```

3. Create a TfidfVectorizer object (let's call it C) with the settings:

   ```
   max_features = 10000, # only top 10k by freq

   lowercase = False,

   ngram_range = (1,2),

   min_df=200,  # note: absolute count of docs

   max_df=0.95  # note: % of docs
   ```

4. Using the training data `documents_train` , along with the ground truth labels `labels_train` , train three Naive Bayes classifiers, corresponding to choices A, B, and C of vectorizer.
5. Normally we'd compute the accuracy of these classifiers on a test set, but for this question we're interested more in the potential upper bound on performance that is achievable with text representation choices A, B, or C. Thus you should compute, for each of the three classifiers, the accuracy on the *training set*.
6. Your function should return these three accuracy scores as a tuple with three floats: (accuracy_A, accuracy_B, accuracy_C).

It is instructive to examine the difference in accuracy across the three different representations.

In [4]:
```
from sklearn.naive_bayes import MultinomialNB
```

In [5]:
```
A=TfidfVectorizer(max_features=10000,
                lowercase=False,
                ngram_range= (1,2),
                min_df=10,
                max_df=0.95,
                stop_words= 'english')
B=CountVectorizer(max_features= 10000,
                 lowercase=False,
                 ngram_range=(1,2),
                 min_df=10,
                 max_df=0.95,
                 stop_words='english')
C=TfidfVectorizer(max_features=10000,
                lowercase=False,
                ngram_range=(1,2),
                min_df=200,
                max_df=0.95)
```

In [6]:
```
from sklearn.metrics import accuracy_score

A_documents=A.fit_transform(documents_train)
A_NB=MultinomialNB().fit(A_documents, labels_train)
A_preds=A_NB.predict(A_documents)
A_score=accuracy_score(labels_train, A_preds)
```

In [7]:
```
print(A_score)
```

```
0.8352046680222792
```

In [8]:
```
B_documents=B.fit_transform(documents_train)
B_NB=MultinomialNB().fit(B_documents, labels_train)
B_preds=B_NB.predict(B_documents)
B_score=accuracy_score(labels_train, B_preds)
```

In [9]:
```
B_score
```

Out[9]:
```
0.7694279904517726
```

In [10]:
```
C_documents=C.fit_transform(documents_train)
C_NB=MultinomialNB().fit(C_documents, labels_train)
C_preds=C_NB.predict(C_documents)
C_score=accuracy_score(labels_train, C_preds)
```

In [11]:
```
C_score
```

Out[11]:
```
0.5562726549376713
```

In [12]:
```
def answer_text_processing():

    result = (0.8352046680222792,
              0.7694279904517726,
              0.5562726549376713)
    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [13]:
```
stu_ans = answer_text_processing()

assert isinstance(stu_ans, tuple), "Q1: Your function should return a tu
assert len(stu_ans) == 3, "Q1: Your tuple should contain three floats."

for i, item in enumerate(stu_ans):
    assert isinstance(
        item, (float, np.floating)
    ), f"Q1: Your answer at index {i} should be a float number. "

# Some hidden tests

del stu_ans
```

Question 2 (30 points). Latent Semantic Indexing and the vocabulary gap.

One of the original motivations for Latent Semantic Indexing was overcoming the `vocabulary gap` in information retrieval. A query like `economic budget` should match strongly against text like `government spending on the economy` even though they don't have any exact keywords in common.

In this question we'll create a demonstration of the power of Latent Semantic Indexing to do semantic matching. In the first part, you'll run LSI and use the reduced document matrix to do semantic matching of a query against other text that has no terms explicitly in common.

In the second part, you'll see how this semantic matching is happening by computing the related terms that are included a query expanded using LSI's latent topics.

Part 2.1 (15 points) Use the reduced document matrix from LSI to do semantic matching of a query against a document.

As a first step, run the code below that we've provided that creates a tf.idf vectorizer and applies it to the 20newsgroups training set. It also runs LSI (in reality a TruncatedSVD) with a latent space of 200 dimensions.

Suppose we have a query "economic budget" that has the tf.idf vector q, with shape 1 x num_terms. We can obtain this vector simply by using vectorizer.transform on the text. Think of the matrix U_k as the super operator that converts from original term space to latent semantic space. To expand text q with related terms according to LSI, compute the expanded query q_k using the formula

$$q_k = q \cdot (U_k \Sigma_k^{-1})$$

With this formula, you'll "expand" both the query and the document vectors to add related terms, and then compute the similarity match between them.

Let's walk through these steps (**Note** that in matrix math dimensions and hence order of operations matters!).

1. The *reduced term matrix*, U_k , is *multiplied* (* operator) with the inverse of matrix Σ_k , the LSI.singular_values_ matrix. **Note** that Σ_k is raised to the power of negative one Σ_k^{-1} . The reason we invert Σ_k is that there is no division operation with matrices so we invert and multiply!
   - Think of this step as forming a normalized scaler for the LSI latent factor weights (the 'topics').
2. The vectorized query matrix q (or document d) is then dotted (dot-product) with the result of $U_k \Sigma_k^{-1}$.

For this question, use cosine similarity to compute the similarity match between any two pieces of text, no matter what their vector representation.

With the formula above, consider the query `"economic budget"` being matched against the (very) short document `"government spending on the economy"` .

Your function should return a tuple of two floats: the cosine similarity score (from sklearn.metrics.pairwise) of (a) the original query and document vectors and (b) the LSI-expanded query and document vectors using the method above.

Did LSI help overcome the vocabulary gap?

In [14]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

### Run this preamble code to run LSI. We've also given the line of code
tfidf_vectorizer = TfidfVectorizer(
    ngram_range=(1, 1), min_df=2, max_df=0.95, stop_words="english", max_
)  # default English stopwords

tfidf_documents = tfidf_vectorizer.fit_transform(documents_train)
tfidf_feature_names = tfidf_vectorizer.get_feature_names()

# LSI does truncated SVD on the document-term matrix of tf.idf term-weig
# The matrix we got back from the vectorizer is a
# document-term matrix, i.e. one row per document.
n_topics = 200
lsi = TruncatedSVD(n_components=n_topics, random_state=0)

# To match the examples and development of LSI in
# our lectures, we're going to
# take the transpose of the document-term matrix to give
# TruncatedSVD the term-document matrix as input.

# This is the matrix U_k:  num_term_features x num_topics
reduced_term_matrix = lsi.fit_transform(np.transpose(tfidf_documents))
```

In [15]:
```
singular_values=[]
for value in lsi.singular_values_:
    singular_values.append(value**-1)
singular_values=np.asarray(singular_values)
```

In [16]:
```
query=["economic budget"]
document=["government spending on the economy"]

query_vector=tfidf_vectorizer.transform(query)
print(query_vector.shape)
document_vector=tfidf_vectorizer.transform(document)
print(document_vector.shape)
```

```
(1, 10000)
(1, 10000)
```

In [17]:
```
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarity(query_vector.toarray(), document_vector.toarray())
```

Out[17]:
```
array([[0.]])
```

In [18]:
```
reduced_term_matrix.shape
```

Out[18]:
```
(10000, 200)
```

In [19]:
```
query_dot=np.dot(query_vector.toarray().astype(float), reduced_term_matr
query_dot
```

Out[19]:
```
array([[ 0.127,  0.034, -0.031, -0.064,  0.06 ,  0.038, -0.017, -0.
,
        -0.048, -0.003,  0.031, -0.003, -0.02 ,  0.05 , -0.045, -0.08
,
        -0.007,  0.037,  0.035,  0.009,  0.053, -0.021,  0.003,  0.027
,
         0.011,  0.044, -0.012,  0.01 , -0.024, -0.026,  0.018,  0.037
,
        -0.018, -0.023,  0.004, -0.03 , -0.032, -0.001, -0.052, -0.008
,
        -0.013,  0.01 , -0.023, -0.005, -0.01 ,  0.005, -0.041,  0.003
,
         0.002,  0.014, -0.019,  0.037,  0.003,  0.019,  0.042, -0.014
,
         0.014,  0.021, -0.01 ,  0.01 ,  0.003, -0.056,  0.019, -0.016
,
        -0.022,  0.017, -0.02 , -0.005, -0.015, -0.012, -0.031,  0.014
,
         0.01 ,  0.007,  0.021,  0.001, -0.   ,  0.005,  0.026, -0.007
,
         0.008, -0.024, -0.005,  0.006, -0.041,  0.01 ,  0.023, -0.02
,
        -0.003,  0.011,  0.004, -0.07 , -0.005, -0.007,  0.001, -0.019
,
         0.002, -0.007, -0.001,  0.01 ,  0.016,  0.018,  0.023,  0.013
,
        -0.004,  0.007, -0.018, -0.021,  0.006, -0.022, -0.002, -0.011
,
         0.001,  0.001, -0.019,  0.027,  0.005, -0.031,  0.001, -0.007
,
        -0.026, -0.009, -0.005, -0.022, -0.01 , -0.007,  0.01 , -0.031
,
         0.023, -0.018, -0.005,  0.042,  0.014, -0.038,  0.006,  0.007
,
         0.009,  0.018,  0.021, -0.014, -0.01 ,  0.015,  0.029,  0.015
,
        -0.016,  0.013,  0.032, -0.001,  0.004,  0.006, -0.007, -0.001
,
        -0.021,  0.006,  0.02 , -0.001,  0.005,  0.022, -0.013, -0.018
,
         0.017, -0.005,  0.016,  0.025, -0.011, -0.019,  0.007, -0.004
,
        -0.015, -0.001,  0.014, -0.014,  0.004,  0.016, -0.009, -0.021
,
          0.016, -0.004,  0.049, -0.004, -0.021, -0.022,  0.023, -0.014
,
         -0.006,  0.    , -0.    ,  0.004,  0.023, -0.001,  0.002, -0.003
,
         -0.004, -0.009, -0.013,  0.023,  0.037,  0.007, -0.005,  0.019
]])
```

In [20]:
```
document_dot=np.dot(document_vector.toarray(), reduced_term_matrix)
document_dot.shape
```

Out[20]:
```
(1, 200)
```

In [21]:
```
from scipy import spatial
query_expanded=singular_values*query_dot
document_expanded=singular_values*document_dot
cos_a=1-spatial.distance.cosine(query_expanded, document_expanded)
cos_a
```

Out[21]:
```
0.3921621600008791
```

In [22]:
```
def answer_semantic_similarity_a():

    result = (0.0, 0.3921621600008791)

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [23]:
```
stu_ans = answer_semantic_similarity_a()

assert isinstance(stu_ans, tuple), "Q2a: Your function should return a tu
assert len(stu_ans) == 2, "Q2a: Your tuple should contain two floats."

for i, item in enumerate(stu_ans):
    assert isinstance(
        item, (float, np.floating)
    ), f"Q2a: Your answer at index {i} should be a float number. "

# Some hidden tests

del stu_ans
```

Part 2.2 (15 points): We want to understand this semantic matching ability a bit more: what terms does LSI think are similar?

To understand why the LSI-expanded vectors get the results they do, we're going to look at what the operator U does to text. In particular, the term-term matrix UU^T tells us the term expansion behavior of this LSI model. Think of the term-term matrix like an operator that first maps a term to the latent space L_k (using U), then back again from L_k to term space (using U transpose). The (i, j) entry of UU^T is a kind of *association weight* between term i and term j.

Write a function to get the most related terms (according to LSI) for the word "economy". To do this:

1. Compute the term-term matrix from the matrix U (the reduced_term_matrix variable).
2. Use the term-term matrix to get the association weights of all words related to the term "economy"
3. Sort by descending weight value.
4. Your function should return the top 5 words and their weights as a list of (string, float) tuples.

Do the related terms match your subjective similarity judgment?

In [24]:
```
term_term_matrix=np.dot(reduced_term_matrix, np.transpose(reduced_term_ma
```

In [25]:
```
top=5
term_index = tfidf_vectorizer.vocabulary_['economy']
top_related_term_indexes = term_term_matrix[term_index, :].argsort()[::-

for i in range(0,top):
    this_term = top_related_term_indexes[i]
    print('\t{} ({})'.format(tfidf_feature_names[this_term], term_term_ma
```

```
	government (0.41470881787021674)
	people (0.258814575680297)
	clinton (0.22583573791646555)
	money (0.20222045017000606)
	president (0.18232552387601855)
```

In [26]:
```
def answer_semantic_similarity_b():

    result = [('government', 0.414708817870216474),
              ('people', 0.258814575680297),
              ('clinton', 0.22583573791646555),
              ('money', 0.20222045017000606),
              ('president', 0.18232552387601855)]

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [27]:
```
stu_ans = answer_semantic_similarity_b()

assert isinstance(stu_ans, list), "Q2b: Your function should return a lis
assert (
    len(stu_ans) == 5
), "Q2b: Your list should contain five elements (the term, score tuples)

for i, item in enumerate(stu_ans):
    assert isinstance(item, tuple), f"Q2b: Your answer at index {i} shoul
    assert isinstance(
        item[0], str
    ), f"Q2b: The first element of your tuple at index {i} should be a st
    assert isinstance(
        item[1], (float, np.floating)
    ), f"Q2b: The second element of your tuple at index {i} should be a f

# Some hidden tests

del stu_ans
```

Question 3 (20 points) Semantic similarity: comparing your ranking with word2vec's ranking

Part 3.1 Ranking words by yourself and then generating word2vec's ranking

First, rearrange the words in the `my_ranking` tuple based on your subjective impression of how similar they are to the word `"party"`. Encode your impression with words arranged in decreasing similarity to `"party"`; index 0 holds `"party"`, index 1 the next most similar term all the way to index 8 holding what you consider the least similar. **This is your personal subjective understanding of how similar these terms are to `"party"`.**

In [28]:
```
# EDIT THE FOLLOWING variable my_ranking.
#
# The target word is **'party'**, which you should keep first in the tup
# in my_ranking so that it reflects
# YOUR subjective ranking for how semantically similar each word is to t
# if you think 'event' is the most similar word to 'party', it should be
# and so on. Make sure you use all the words : just re-order them.
my_ranking = ('party', 'event', 'fun', 'vote', 'election',  'lead', 'char
```

Before proceeding:

- The following code cell must be executed to load the pre-trained word2vec model.

In [29]:
```
#### We need to load the pre-trained word2vec model.
#### The result is an instance of the class W2VTransformer(size=100, min_
#### from gensim.sklearn_api import W2VTransformer

import pickle

f = open("./assets/text8_W2V.pickle", "rb")
text8_model = pickle.load(f)
f.close()
```

The second part of the task is to derive the model's understanding of how similar these terms are to the string "party". One way to do this:

1. Use the word2vec `text8_model` to convert `my_ranking` into the corresponding set of word embedding vectors.
2. Determine the cosine similarity of each word embedding relative to the target embedding of `"party"`. Maintain a list of tuples in the format of `(similarity_score, word)`. Do not skip over the embedding for the target word `"party"`, the list should include the similarity to itself.
3. Sort the list in decreasing order. Remember, when sorting elements of type tuple, the first element determines the sorting order and the second element is only considered if there is a tie.
4. Create a list variable called `system_ranking` which contains only the second element of each tuple (the words).

Your function should return a tuple in the format of `(my_ranking, system_ranking)`.

In [30]:
```
system_ranking = text8_model.transform(my_ranking)
```

In [31]:
```
from sklearn.metrics.pairwise import cosine_similarity
for i in range(1, len(system_ranking)):
    print(cosine_similarity([system_ranking[0]], [system_ranking[i]]))
```

```
[[0.204]]
[[-0.011]]
[[0.568]]
[[0.653]]
[[0.135]]
[[0.045]]
[[0.35]]
[[0.001]]
```

In [32]:
```
system_order = ('party', 'election', 'vote', 'budget', 'event', 'lead',
```

In [33]:
```
from scipy.stats import spearmanr
spearmanr_ranking = spearmanr(my_ranking, system_order)
```

In [34]:
```
spearmanr_ranking
```

Out[34]:
```
SpearmanrResult(correlation=0.2833333333333333, pvalue=0.4600303289657
1994)
```

In [35]:
```
def answer_word2vec1():
    result = (my_ranking, system_order, spearmanr_ranking)

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [36]:
```
reference_terms = (
    "party",
    "bicycle",
    "vote",
    "lead",
    "election",
    "champagne",
    "event",
    "fun",
    "budget",
)

stu_ans = answer_word2vec1()

assert isinstance(stu_ans, tuple), "Q3.1: Your function should return a t
assert (
    len(stu ans) == 2
    len(stu_ans) == 2
), "Q3.1: Your tuple should contain two elements: a tuple of strings (my_

# check my_rankings
assert isinstance(
    stu_ans[0], tuple
), "Q3.1: Your first element must be a tuple (of strings). "
assert len(stu_ans[0]) == len(
    reference_terms
), "Q3.1: Your my_rankings tuple doesn't have the expected number of terr
assert (
    stu_ans[0][0] == reference_terms[0]
), "Q3.1: Your my_rankings tuple must have 'party' as the first term."
assert set(stu_ans[0]) == set(
    reference_terms
), "Q3.1: Your my_rankings tuple is not a permutation of the permitted te

# check system_rankings
assert isinstance(
    stu_ans[1], tuple
), "Q3.1: Your second element must be a tuple (of strings). "
assert len(stu_ans[1]) == len(
    reference_terms
), "Q3.1: Your system_rankings tuple doesn't have the expected number of
assert (
    stu_ans[0][0] == reference_terms[0]
), "Q3.1: Your system_rankings tuple must have 'party' as the first term
assert set(stu_ans[1]) == set(
    reference_terms
), "Q3.1: Your system_rankings tuple is not a permutation of the permitte

# Some hidden tests

del stu_ans
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call
last)
Input In [36], in <cell line: 17>()
     13 stu_ans = answer_word2vec1()
     15 assert isinstance(stu_ans, tuple), "Q3.1: Your function should
return a tuple. "
---> 16 assert (
     17     len(stu_ans) == 2
     18 ), "Q3.1: Your tuple should contain two elements: a tuple of s
trings (my_ranking), and a tuple of strings (system_ranking)."
     20 # check my_rankings
     21 assert isinstance(
     22     stu_ans[0], tuple
     23 ), "Q3.1: Your first element must be a tuple (of strings). "
```

```
AssertionError: Q3.1: Your tuple should contain two elements: a tuple
of strings (my_ranking), and a tuple of strings (system_ranking).
```

Part 3.2 Comparing rankings with Spearman correlation coefficient

Given the rankings you generated above in the format of `(my_ranking, system_ranking)` , use the [`scipy.stats.spearmanr` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html) function which will return a SciPy object containing the spearman correlation and the p-value of your ranking compared to the system ranking, as a measurement of how well they agree. Convert this spearman result to a tuple, and append that tuple to your previous results, i.e. it should have a format of `(my_ranking, system_ranking, spearman_tuple)` .

In [37]:
```
def answer_word2vec():
    result = (my_ranking, system_order, spearmanr_ranking)

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [38]:
```
reference_terms = (
    "party",
    "bicycle",
    "vote",
    "lead",
    "election",
    "champagne",
    "event",
    "fun",
    "budget",
)

stu_ans = answer_word2vec()

assert isinstance(stu_ans, tuple), "Q3.2: Your function should return a t
assert (
    len(stu_ans) == 3
), "Q3.2: Your tuple should contain three elements: a tuple of strings (r

# check my_rankings
assert isinstance(
    stu_ans[0], tuple
), "Q3.2: Your first element must be a tuple (of strings). "
```

```
assert len(stu_ans[0]) == len(
    reference_terms
), "Q3.2: Your my_rankings tuple doesn't have the expected number of terr
assert (
    stu_ans[0][0] == reference_terms[0]
), "Q3.2: Your my_rankings tuple must have 'party' as the first term."
assert set(stu_ans[0]) == set(
    reference_terms
), "Q3.2: Your my_rankings tuple is not a permutation of the permitted te

# check system_rankings
assert isinstance(
    stu_ans[1], tuple
), "Q3.2: Your second element must be a tuple (of strings). "
assert len(stu_ans[1]) == len(
    reference_terms
), "Q3.2: Your system_rankings tuple doesn't have the expected number of
assert (
    stu_ans[0][0] == reference_terms[0]
), "Q3.2: Your system_rankings tuple must have 'party' as the first term
assert set(stu_ans[1]) == set(
    reference_terms
), "Q3.2: Your system_rankings tuple is not a permutation of the permitte

# check spearmanr
assert isinstance(
    stu_ans[2], tuple
), "Q3.2: Your third element must be a tuple (of two floats). "
assert (
    len(stu_ans[2]) == 2
), "Q3.2: Your spearman output tuple should contain two floats."
assert isinstance(
    stu_ans[2][0], (float, np.floating)
), "Q3.2: Your spearman corr should be a float. "
assert isinstance(
    stu_ans[2][1], (float, np.floating)
), "Q3.2: Your spearman p-val should be a float. "


# Some hidden tests

del stu_ans
```

Question 4: (30 points) Topic coherence.

One measure of topic model quality that is used e.g. to determine the optimal number of topics for a corpus is *topic coherence*. This is a measure of how semantically related the top terms in a topic model are. Topic models with low coherence tend to be filled with seemingly random words and hard to interpret, while high coherence usually indicates a clear semantic theme that's easily understood.

With their ability to represent word semantics, word embeddings are an ideal tool for computing topic coherence. In part 1, you'll implement a simple topic coherence function. In part 2, you'll apply that function to NMF topic modeling to find a setting for the number of topics that gives maximally coherent topic models.

We're going to use the same `text8_model` W2VTransformer object, which implements the word2vec embedding, that you loaded for the previous question.

Part 4.1. (15 points) Average semantic distance as a text coherence measure.

Implement a function that takes a list of terms (strings) as input and returns a positive float indicating their semantic coherence. Here is the algorithm you should use:

1. For each input term, compute its word2vec embedding vector. One problem you might encounter is that some terms may not exist in the word2vec model. You get a "KeyError" exception when trying to transform that "out-of-vocabulary" term. You should ignore these terms: one way to do this is by wrapping your embedding call with a try/except statement that catches the KeyError and just ignores that word, and continues processing.
2. Once you have the list of embedding vectors for the input terms, compute their pairwise cosine similarity. If there are n embedding vectors, this step will result in an nxn matrix D. If for some reason there are no input terms remaining (they are all out-of-vocabulary) just return 0.
3. Obviously the most similar word to a term is itself, indicated by a "1" on the diagonal of D. But we don't want those: we only care about the pairwise distances to *other* terms, so to deal that case, set the diagonal to zero.
4. Return the mean over all pairwise distances in D (with self-distances set to zero). This is our simple coherence measure.

Be sure to try it out on some samples. For example, here's what our reference implementation returns:

```
topical_coherence(['car', 'airplane', 'taxi', 'bus', 'vehicle',
'transport'])
```

0.46063321000999874

`topical_coherence(['apple', 'banana', 'cherry', 'watermelon', 'lemon', 'orange'])`

0.43306025200419956

`topical_coherence(['possible', 'mean', 'volcano', 'feature', 'record', 'quickly'])`

0.1150558124192887

Your function should return the above measure of topic coherence for the following three lists, as a tuple of three corresponding floats:

`['train', 'car', 'bicycle', 'bus', 'vehicle', 'transport']`

`['scsi', 'drive', 'computer', 'storage', 'megabyte']`

`['introduction', 'pickle', 'guard', 'red', 'valiant']`

In [39]:
```python
test=['apple', 'banana', 'cherry', 'watermelon', 'lemon', 'orange']
vecs=[]
for word in test:
    try:
        vecs.append(text8_model.transform(word))
    except KeyError:
        pass
```

In [40]:
```python
vecs=text8_model.transform(test)
```

In [41]:
```python
temp=cosine_similarity(vecs)
for i in range(len(temp)):
    for j in range(len(temp)):
        if i == j:
            temp[i][j]=0
temp
```

Out[41]:
```
array([[0.   , 0.273, 0.163, 0.155, 0.234, 0.181],
       [0.273, 0.   , 0.788, 0.629, 0.758, 0.579],
       [0.163, 0.788, 0.   , 0.758, 0.867, 0.659],
       [0.155, 0.629, 0.758, 0.   , 0.702, 0.398],
       [0.234, 0.758, 0.867, 0.702, 0.   , 0.652],
       [0.181, 0.579, 0.659, 0.398, 0.652, 0.   ]], dtype=float32)
```

In [42]:
```
np.mean(temp)
```

Out[42]:
```
0.43306026
```

In [43]:
```
def answer_coherence_a():
    result = (0.5008174, 0.44538254, 0.10494587)

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [44]:
```
stu_ans = answer_coherence_a()

assert isinstance(stu_ans, tuple), "Q4.1: Your function should return a t
assert (
    len(stu_ans) == 3
), "Q4.1: Your function should return a tuple of three elements. "

for i, item in enumerate(stu_ans):
    assert isinstance(
        item, (float, np.floating)
    ), f"Q4.1: Your answer at index {i} should be a float number. "


# Some hidden tests

del stu_ans
```

Part 4.2 (15 points) Applying semantic coherence to topic model selection.

Now you'll use the semantic coherence measure you developed in Part 1 with topic models computed using Non-Negative Matrix Factorization.

Implement a simple loop that trains an NMF topic model, for number of topics **from 2 to 10 inclusive**. At each iteration, compute your topic coherence measure on the **top 10** words for each topic. Then compute the *median* topic coherence over all these topic scores.

Your function should return a list of 9 median coherence scores, corresponding to each choice of the number of topics to use with NMF. Which choice gives the highest median semantic coherence?

When creating the NMF object, use these parameter settings: `random_state=42, init="nndsvd"` .

In [59]:
```
### Use the following code to prepare input to the NMF topic model.
### It assumes you've loaded the 20newgroups variables at the beginning o
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer_NMF = TfidfVectorizer(
    max_features=20000,  # only top 5k by freq
    lowercase=True,  # drop capitalization
    ngram_range=(1, 1),
    min_df=2,  # note: absolute count of doc
    max_df=0.05,  # note: % of docs
    token_pattern=r"\b[a-z]{3,12}\b",  # remove short, non-word-like tern
    stop_words="english",
)  # default English stopwords

tfidf_documents_NMF = tfidf_vectorizer_NMF.fit_transform(documents_train
feature_names_NMF = tfidf_vectorizer_NMF.get_feature_names()
```

In [60]:
```
feature_names_NMF
```

Out[60]:
```
['aaa',
 'aamir',
 'aaron',
 'aas',
 'abandon',
 'abandoned',
 'abbey',
 'abbott',
 'abbreviation',
 'abc',
 'abd',
 'abdomen',
 'abdominal',
 'abdullah',
 'aberdeen',
 'abhor',
 'abide',
 'abiding',
 'abilities',
```

In [62]:
```
def get_med(n_component):
    np.random.seed(42)
    from sklearn.decomposition import NMF
    nmf = NMF(
        n_components = n_component,
        random_state = 42,
        init='nndsvd').fit(tfidf_documents_NMF)
    all_topics = []
    for topic_idx, topic in enumerate(nmf.components_):
        top_features_ind = np.argsort(-topic)[:10]
        top_features = [feature_names_NMF[i] for i in top_features_ind]
        all_topics.append(top_features)
    topcos = []
    for word in all_topics:
        top_co = topic_coherence(word)
        topcos.append(top_co)
    return np.median(topcos)
```

```
  Input In [62]
    init='nndsvd').fit(tfidf_documents_NMF)
                ^
SyntaxError: invalid character in identifier
```

In [63]:
```
def answer_coherence_b():
    result = [('2.0'),('3.0'),('4.0'),('5.0'),('6.0'), ('7.0'), ('8.0'),

    # YOUR CODE HERE
    #raise NotImplementedError()

    return result
```

In [64]:
```
stu_ans = answer_coherence_b()

assert isinstance(stu_ans, list), "Q4.2: Your function should return a li
assert (
    len(stu_ans) == 9
), "Q4.2: Your function should return a list of nine elements (topic coun

for i, item in enumerate(stu_ans):
    assert isinstance(
        item, (float, np.floating)
    ), f"Q4.2: Your answer at index {i} should be a float number. "


# Some hidden tests

del stu_ans
```

```
---------------------------------------------------------------------------
-----
AssertionError                            Traceback (most recent call
last)
Input In [64], in <cell line: 8>()
      4 assert (
      5     len(stu_ans) == 9
      6 ), "Q4.2: Your function should return a list of nine elements
(topic count 2 thru 10). "
      8 for i, item in enumerate(stu_ans):
----> 9     assert isinstance(
     10         item, (float, np.floating)
     11     ), f"Q4.2: Your answer at index {i} should be a float numb
er. "
     14 # Some hidden tests
     16 del stu_ans

AssertionError: Q4.2: Your answer at index 0 should be a float number.
```

In [ ]: