L4: Word embeddings

In this lab you will explore word embeddings. A **word embedding** is a mapping of words to points in a vector space such that nearby words (points) are similar in terms of their distributional properties. You will use word embedding to find similar words, and evaluate their usefulness in an inference task.

You will use the word vectors that come with spaCy. Note that you will need the 'large' English language model; the 'small' model that you used in previous labs does not include proper word vectors.

Students:

Weng Hang Wong (wenwo535)

Zuxiang Li(zuxli371)

In [49]:
```
import spacy

nlp = spacy.load('en_core_web_lg')
```

Every word in the model's vocabulary comes with a 300-dimensional vector, represented as a NumPy array. The following code cell shows how to access the vector for the word *cheese*:

Problem 1: Finding similar words

Your first task is to use the word embeddings to find similar words. More specifically, we ask you to write a function `most_similar` that takes a vector x and returns a list with the 10 most similar entries in spaCy's vocabulary, with similarity being defined by cosine.

Tip: spaCy already has a `most_similar` method that you can wrap.

In [50]:
```
import numpy as np
# TODO: Enter your implementation of `most_similar` here
def most_similar(vector,n=10):
    #vector=nlp.vocab['cheese'].vector
    query=vector.reshape((1,vector.shape[0]))
    res=nlp.vocab.vectors.most_similar(query,n=n)
    return [nlp.vocab.strings[key] for key in res[0][0]]
```

Test your implementation by running the following code cell, which will print the 10 most similar words for the word *cheese*:

In [51]:
```
print(' '.join(w for w in most_similar(nlp.vocab['cheese'].vector)))
```

```
CHEESE cheese Cheese Cheddar cheddar CHEDDAR BACON Bacon bacon cheeses
```

You should get the following output:

CHEESE cheese Cheese Cheddar cheddar CHEDDAR BACON Bacon bacon cheeses

Once you have a working implementation of `most_similar`, use it to think about in what sense the returned words really are 'similar' to the cue word. Try to find examples where the cue word and at least one of the words returned by `most_similar` are in the following semantic relations:

1. synonymy (exchangeable meanings)
2. antonymy (opposite meanings)
3. hyperonymy/hyponymy (more specific/less specific meanings)

Document your examples in the code cell below.

In [52]:
```
# TODO: Insert code here to generate your examples
most_similar(nlp.vocab["cold"].vector)
```

Out[52]:
```
['cOLD',
 'COLD',
 'Cold',
 'cold',
 'Chilly',
 'CHILLY',
 'chilly',
 'WARM',
 'Warm',
 'warm']
```

Problem 2: Plotting similar words

Your next task is to visualize the word embedding space by a plot. To do so, you will have to reduce the dimensionality of the space from 300 to 2 dimensions. One suitable algorithm for this is T-distributed Stochastic Neighbor Embedding (TSNE), which is implemented in scikit-learn's TSNE class.

Write a function `plot_most_similar` that takes a list of words (lexemes) and does the following:

1. For each word in the list, find the most similar words (lexemes) in the spaCy vocabulary.
2. Compute the TSNE transformation of the corresponding vectors to 2 dimensions.
3. Produce a scatter plot of the transformed vectors, with the vectors as points and the corresponding word forms as labels.

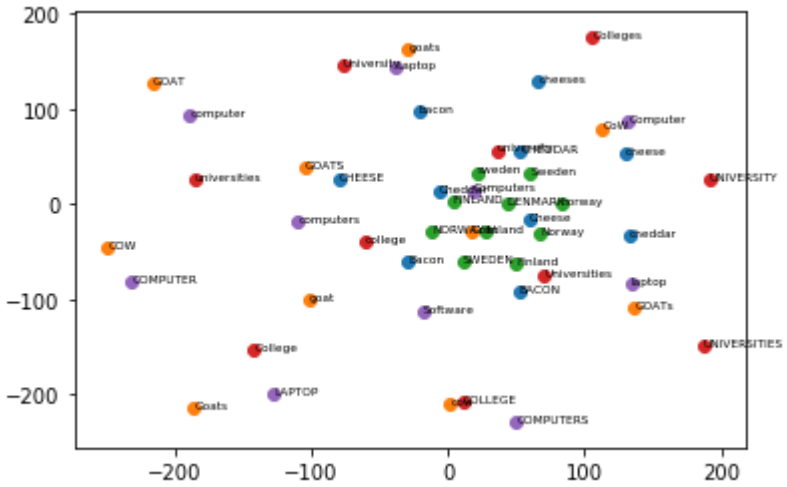
In [53]:
```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
# TODO: Write code here to plot the most similar words
def plot_most_similar(words):
    for i,word in enumerate(words):
        #similar_words=most_similar(nlp.vocab[word].vector,n=10)
        similar_words=most_similar(word.vector,n=10)
        print(similar_words)
        vectors=[]
        for w in similar_words:
            vectors.append(nlp.vocab[w].vector.reshape((1,300)))
        all_vecs=np.concatenate(vectors,axis=1).reshape(10,300)
        tsne=TSNE(n_components=2).fit_transform(all_vecs)
        x,y=[x[0] for x in tsne],[x[1] for x in tsne]
        #plt.figure()
        plt.scatter(x=x,y=y)
        for j in range(10):
            plt.annotate(similar_words[j],(x[j],y[j]),size="xx-small")

#plot_most_similar(["university","sweden"])
```

Test your code by running the following cell:

In [54]:
```
plot_most_similar(nlp.vocab[w] for w in ['cheese', 'goat', 'sweden', 'univer
```

```
['CHEESE', 'cheese', 'Cheese', 'Cheddar', 'cheddar', 'CHEDDAR', 'BACON', 'Ba
con', 'bacon', 'cheeses']
['goat', 'Goat', 'GOAT', 'goats', 'GOATS', 'Goats', 'GOATs', 'COW', 'CoW', '
cow']
['sweden', 'Sweden', 'SWEDEN', 'Finland', 'FINLAND', 'finland', 'NORWAY', 'n
orway', 'Norway', 'DENMARK']
['University', 'UNIVERSITY', 'university', 'COLLEGE', 'College', 'college',
'Universities', 'universities', 'UNIVERSITIES', 'Colleges']
['computer', 'COMPUTER', 'Computer', 'computers', 'COMPUTERS', 'Computers',
'LAPTOP', 'laptop', 'Laptop', 'Software']
```



Take a few minutes to look at your plot. What does it tell you? What does it *not* tell you?

The plot above showed the distance between words that are similar, but it didn't show the distance between words in different normal norm, since everytime we run this algorithm it returns

similar pattern between words that are similar but different in words that are not similar

Problem 3: Analogies

In a **word analogy task** you are given three words x, y, z and have to predict a word w that has the same semantic relation to z as y has to x. One example is *man*, *woman*, *brother*, the expected answer being *sister* (the semantic relation is *male*/*female*).

Mikolov et al. (2013) have shown that some types of word analogy tasks can be solved by adding and substracting word vectors in a word embedding: the vector for *sister* is the closest vector (in terms of cosine distance) to the vector *brother* $-$ *man* $+$ *woman*. Your next task is to write a function `fourth` that takes in three words (say *brother*, *man*, *woman*) and predicts the word that completes the analogy (in this case, *sister*).

In [11]:
```
# TODO: Enter code here to solve the analogy problem

def fourth(word1,word2,word3):
    return most_similar(word1.vector-word2.vector+word3.vector)[0]
```

Test your code by running the following code. You should get *sister*.

In [12]:
```
print(fourth(nlp.vocab['brother'], nlp.vocab['man'], nlp.vocab['woman']))
print(fourth(nlp.vocab['Stockholm'], nlp.vocab['Sweden'], nlp.vocab["Germany
print(fourth(nlp.vocab['Swedish'], nlp.vocab['Sweden'], nlp.vocab["France"]]
print(fourth(nlp.vocab['better'], nlp.vocab['good'], nlp.vocab["bad"]))
print(fourth(nlp.vocab['walked'], nlp.vocab['walk'], nlp.vocab["take"]))
```

```
SISTER
BERLIN
French
WORSE
TOOK
```

In [25]:
```
print(fourth(nlp.vocab['walked'], nlp.vocab['walk'], nlp.vocab['run']))
print(fourth(nlp.vocab['chinese'], nlp.vocab['china'], nlp.vocab['japanese']
print(fourth(nlp.vocab['beijing'], nlp.vocab['china'], nlp.vocab['japan']))

print(fourth(nlp.vocab['chinese'], nlp.vocab['china'], nlp.vocab['france']))
print(fourth(nlp.vocab['japanese'], nlp.vocab['japan'], nlp.vocab['france']]
print(fourth(nlp.vocab['tokyo'], nlp.vocab['japan'], nlp.vocab['Germany']))
```

```
RAN
japanese
TOKYO
france
france
Germany
```

You should also be able to get the following:

- *Stockholm* $-$ *Sweden* $+$ *Germany* $=$ *Berlin*
- *Swedish* $-$ *Sweden* $+$ *France* $=$ *French*
- *better* $-$ *good* $+$ *bad* $=$ *worse*

- *walked* − *walk* + *take* = *took*

Experiment with other examples to see whether you get the expected output. Provide three examples of analogies for which the model produces the 'correct' answer, and three examples on which the model 'failed'. Based on your theoretical understanding of word embeddings, do you have a hypothesis as to why the model succeeds/fails in completing the analogy? Discuss this question in a short text.

TODO: Insert your examples and your discussion here

After reading the article above we can know for our successful examples here, all pair of words sharing aparticular relation are related by the same constant offset. As for the failures, our guess is that the pair of words doesn't occur in a same embedding space or the distance between them is too far.

Natural language inference dataset

In the second part of this lab, you will be evaluating the usefulness of word embeddings in the context of a natural language inference task. The data for this part is the SNLI corpus, a collection of 570k human-written English image caption pairs manually labeled with the labels *Entailment*, *Contradiction*, and *Neutral*. Consider the following sentence pair as an example:

- Sentence 1: A soccer game with multiple males playing.
- Sentence 2: Some men are playing a sport.

This pair is labeled with *Entailment*, because sentence 2 is logically entailed (implied) by sentence 1 – if sentence 1 is true, then sentence 2 is true, too. The following sentence pair, on the other hand, is labeled with *Contradiction*, because both sentences cannot be true at the same time.

- Sentence 1: A black race car starts up in front of a crowd of people.
- Sentence 2: A man is driving down a lonely road.

For detailed information about the corpus, refer to Bowman et al. (2015). For this lab, we load the training portion and the development portion of the dataset.

Note: Because the SNLI corpus is rather big, we initially only load a small portion (25,000 samples) of the training data. Once you have working code for Problems 4–6, you should set the flag `final` to `True` and re-run all cells with the full dataset.

In [27]:
```
import bz2
import pandas as pd

final_evaluation = False    # TODO: Set to True for the final evaluation!

with bz2.open('train.jsonl.bz2', 'rt') as source:
    if final_evaluation:
        df_train = pd.read_json(source, lines=True)
    else:
        df_train = pd.read_json(source, lines=True, nrows=25000)
    print('Number of sentence pairs in the training data:', len(df_train))

with bz2.open('dev.jsonl.bz2', 'rt') as source:
    df_dev = pd.read_json(source, lines=True)
    print('Number of sentence pairs in the development data:', len(df_dev))
```

```
Number of sentence pairs in the training data: 25000
Number of sentence pairs in the development data: 9842
```

When you inspect the data frames, you will see that we have preprocessed the sentences and separated tokens by spaces. In the columns `tagged1` and `tagged2`, we have added the part-of-speech tags for every token (as predicted by spaCy), also separated by spaces.

In [10]:
```
df_train.head()
```

Out[10]:

	gold_label	sentence1	tags1	sentence2	tags2
0	neutral	A person on a horse jumps over a broken down a...	DET NOUN ADP DET NOUN VERB ADP DET ADJ ADP NOU...	A person is training his horse for a competiti...	DET NOUN AUX VERB PRON NOUN ADP DET NOUN PUNCT
1	contradiction	A person on a horse jumps over a broken down a...	DET NOUN ADP DET NOUN VERB ADP DET ADJ ADP NOU...	A person is at a diner , ordering an omelette .	DET NOUN AUX ADP DET NOUN PUNCT VERB DET NOUN ...
2	entailment	A person on a horse jumps over a broken down a...	DET NOUN ADP DET NOUN VERB ADP DET ADJ ADP NOU...	A person is outdoors , on a horse .	DET NOUN AUX ADV PUNCT ADP DET NOUN PUNCT
3	neutral	Children smiling and waving at camera	NOUN VERB CCONJ VERB ADP NOUN	They are smiling at their parents	PRON AUX VERB ADP PRON NOUN
4	entailment	Children smiling and waving at camera	NOUN VERB CCONJ VERB ADP NOUN	There are children present	PRON AUX NOUN ADJ

Problem 4: Two simple baselines

Your first task is to establish two simple baselines for the natural language inference task.

Random baseline

Implement the standard random baseline that generates prediction by sampling from the empirical distribution of the classes in the training data. Write code to evaluate the performance

of this classifier on the development data.

In [31]:
```
# TODO: Enter code here to implement the random baseline. Print the classifi
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix,plot_confusion_matrix,classific

d=DummyClassifier(strategy="uniform")

d.fit(df_train["sentence1"]+df_train["sentence2"],df_train["gold_label"])
res=d.predict(df_dev["sentence1"]+df_dev["sentence2"])
print(confusion_matrix(df_dev["gold_label"],res))
print(classification_report(df_dev["gold_label"],res))
```

```
[[1070 1115 1093]
 [1130 1119 1080]
 [1074 1091 1070]]
              precision    recall  f1-score   support

contradiction       0.33      0.33      0.33      3278
   entailment       0.34      0.34      0.34      3329
      neutral       0.33      0.33      0.33      3235

     accuracy                           0.33      9842
    macro avg       0.33      0.33      0.33      9842
 weighted avg       0.33      0.33      0.33      9842
```

One-sided baseline

A second obvious baseline for the inference task is to predict the class label of a sentence pair based on the text of only one of the two sentences, just as in a standard document classification task. Put together a simple CountVectorizer + LogisticRegression pipeline that implements this idea, train it, and evaluate it on the development data. Is it better to base predictions on sentence 1 or sentence 2? Why should one sentence be more useful than the other?

In [32]:
```
# TODO: Enter code here to implement the one-sentence baselines. Print the c
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
pipe=Pipeline([("vectorize",CountVectorizer(binary=False,ngram_range=(1,1))]
pipe.fit(df_train["sentence1"],df_train["gold_label"])
res=pipe.predict(df_dev["sentence1"])
print(classification_report(df_dev["gold_label"],res))
print(confusion_matrix(df_dev["gold_label"],res))
pipe.fit(df_train["sentence2"],df_train["gold_label"])
res=pipe.predict(df_dev["sentence2"])
print(confusion_matrix(df_dev["gold_label"],res))
print(classification_report(df_dev["gold_label"],res))
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
```

```
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
              precision    recall  f1-score   support

contradiction       0.33      0.25      0.29      3278
   entailment       0.34      0.34      0.34      3329
      neutral       0.33      0.41      0.36      3235

     accuracy                           0.33      9842
    macro avg       0.33      0.33      0.33      9842
 weighted avg       0.33      0.33      0.33      9842

[[ 825 1116 1337]
 [ 839 1119 1371]
 [ 823 1091 1321]]
[[1963  710  605]
 [ 664 2135  530]
 [ 653  760 1822]]
              precision    recall  f1-score   support

contradiction       0.60      0.60      0.60      3278
   entailment       0.59      0.64      0.62      3329
      neutral       0.62      0.56      0.59      3235

     accuracy                           0.60      9842
    macro avg       0.60      0.60      0.60      9842
 weighted avg       0.60      0.60      0.60      9842
```

TODO: Enter your answer to the discussion questions here

Sentence 2 performs better tha sentence 1. Since in our training data, sentence1 with different gold label owns identical unigrams. That makes it difficult to do classification for our classifier

Problem 5: A classifier based on manually engineered features

Bowman et al., 2015 evaluate a classifier that uses (among others) **cross-unigram features**. This term is used to refer to pairs of unigrams (w_1, w_2) such that w_1 occurs in sentence 1, w_2 occurs in sentence 2, and both have been assigned the same part-of-speech tag.

Your next task is to implement the cross-unigram classifier. To this end, the next cell contains skeleton code for a transformer that you can use as the first component in a classification pipeline. This transformer converts each row of the SNLI data frame into a space-separated

string consisting of

- the standard unigrams (of sentence 1 or sentence 2 – choose whichever performed better in Problem 4)
- the cross-unigrams, as described above.

The space-separated string forms a new 'document' that can be passed to a vectorizer in exactly the same way as a standard sentence in Problem 4.

In [34]:
```
from sklearn.base import BaseEstimator, TransformerMixin

class CrossUnigramsTransformer(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass
    def fit(self, X, y=None):
        return self
    # Transform a single row of the dataframe.
    def _transform(self, row):
        # TODO: Replace the following line with your own code

        s1=row[1].strip().split()
        s2=row[3].strip().split()
        t1=row[2].strip().split()
        t2=row[4].strip().split()
        cu=[]
        for i,w in enumerate(s1):
            if w in s2 and t1[i]==t2[s2.index(w)]:
                cu.append(w)
        return " ".join([w for w in (s2+cu)])

    def transform(self, X):
        return [self._transform(row) for row in X.itertuples()]
```

Once you have an implementation of the transformer, extend the pipeline that you built for Problem 4, train it, and evaluate it on the development data.

In [35]:
```
# TODO: Enter code here to implement the cross-unigrams classifier. Print th

pipe=Pipeline([("transform",CrossUnigramsTransformer()),("vectorize",CountVe
pipe.fit(df_train.drop(["gold_label"],axis=1),df_train["gold_label"])

res=pipe.predict(df_dev.drop(["gold_label"],axis=1))

print(confusion_matrix(df_dev["gold_label"],res))
print(classification_report(df_dev["gold_label"],res))
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
```

```
ssion
  extra warning msg= LOGISTIC SOLVER CONVERGENCE MSG)
[[2023  676  579]
 [ 814 1924  591]
 [ 828  726 1681]]
              precision    recall  f1-score   support

contradiction       0.55      0.62      0.58      3278
   entailment       0.58      0.58      0.58      3329
      neutral       0.59      0.52      0.55      3235

     accuracy                           0.57      9842
    macro avg       0.57      0.57      0.57      9842
 weighted avg       0.57      0.57      0.57      9842
```

Problem 6: A classifier based on word embeddings

Your last task in this lab is to build a classifier for the natural language inference task that uses word embeddings. More specifically, we ask you to implement a vectorizer that represents each sentence as the sum of its word vectors – a representation known as the **continuous bag-of-words**. Thus, given that spaCy's word vectors have 300 dimensions, each sentence will be transformed into a 300-dimensional vector. To represent a sentence pair, the vectorizer should concatenate the vectors for the individual sentences; this yields a 600-dimensional vector. This vector can then be passed to a classifier.

The next code cell contains skeleton code for the vectorizer. You will have to implement two methods: one that maps a single sentence to a vector (of length 300), and one that maps a sentence pair to a vector (of length 600).

In [36]:
```
import numpy as np

from sklearn.base import BaseEstimator, TransformerMixin

class PairedSentenceVectorizer(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    # Vectorize a single sentence.
    def _transform1(self, sentence):
        # TODO: Replace the following line with your own code
        #return np.zeros(nlp.vocab.vectors.shape[1])
        #print(sentence)
        s=sentence.strip().split()
        #print(s)
        #nlp.vocab['cheese'].vector
        init_vector=np.zeros(shape=(1,300))
        for w in s:
            init_vector=init_vector+nlp.vocab[w].vector.reshape(1,300)
        return init_vector
    # Vectorize a single row of the dataframe.
    def _transform2(self, row):
        # TODO: Replace the following line with your own code
        #return np.zeros(nlp.vocab.vectors.shape[1] * 2)
        return np.concatenate((self._transform1(row[1]),self._transform1(row

    def transform(self, X):
        return np.concatenate(
            [self._transform2(row).reshape(1, -1) for row in X.itertuples()]
        )
```

Once you have a working implementation, build a pipeline consisting of the new vectorizer and a multi-layer perceptron classifier. This more powerful (compared to logistic regression) classifier is called for here because we do not specify features by hand (as we did in Problem 5), but want to let the model learn a good representation of the data by itself. Use 3 hidden layers, each with size 300. It suffices to train the classifier for 8 iterations (epochs).

In [37]:
```
# TODO: Enter code here to implement the word embeddings classifier. Print t
from sklearn.neural_network import MLPClassifier

pipe=Pipeline([("vectorize",PairedSentenceVectorizer()),("MLP",MLPClassifier
pipe.fit(df_train.drop(["gold_label"],axis=1),df_train["gold_label"])

res=pipe.predict(df_dev.drop(["gold_label"],axis=1))
print(classification_report(df_dev["gold_label"],res))
print(confusion_matrix(df_dev["gold_label"],res))
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\neura
l_network\_multilayer_perceptron.py:585: ConvergenceWarning: Stochastic Opti
mizer: Maximum iterations (8) reached and the optimization hasn't converged
yet.
  % self.max_iter, ConvergenceWarning)
```

```
                precision    recall  f1-score   support

 contradiction       0.70      0.54      0.61      3278
    entailment       0.65      0.66      0.66      3329
       neutral       0.57      0.69      0.62      3235

      accuracy                           0.63      9842
     macro avg       0.64      0.63      0.63      9842
  weighted avg       0.64      0.63      0.63      9842

[[1774  603  901]
 [ 332 2211  786]
 [ 412  589 2234]]
```

Problem 7: Final evaluation

Once you have working code for all problems, re-run Problems 4–6 with the full training data. This will take quite a while (expect approximately 1&;nbsp;hour on Colab). **Make sure to not overwrite your previous results.** What are your results on the full data? How do they differ from the results that you obtained for the smaller training data? How do you interpret this? Summarize your findings in a short text.

In [38]:

```
# TODO: Enter your code for the full experiments here
final_evaluation =  True   # TODO: Set to True for the final evaluation!

with bz2.open('train.jsonl.bz2', 'rt') as source:
    if final_evaluation:
        df_train = pd.read_json(source, lines=True)
    else:
        df_train = pd.read_json(source, lines=True, nrows=25000)
    print('Number of sentence pairs in the training data:', len(df_train))

with bz2.open('dev.jsonl.bz2', 'rt') as source:
    df_dev = pd.read_json(source, lines=True)
    print('Number of sentence pairs in the development data:', len(df_dev))
```

```
Number of sentence pairs in the training data: 549367
Number of sentence pairs in the development data: 9842
```

In [39]:

```
# baseline
d=DummyClassifier(strategy="uniform")

d.fit(df_train["sentence1"]+df_train["sentence2"],df_train["gold_label"])
d_res=d.predict(df_dev["sentence1"]+df_dev["sentence2"])
```

In [40]:

```
print(confusion_matrix(df_dev["gold_label"],d_res))
print(classification_report(df_dev["gold_label"],d_res))
```

```
[[1110 1098 1070]
 [1134 1103 1092]
 [1085 1111 1039]]
                precision    recall  f1-score   support

 contradiction       0.33      0.34      0.34      3278
    entailment       0.33      0.33      0.33      3329
```

```
     neutral       0.32      0.32      0.32      3235

    accuracy                           0.33      9842
   macro avg       0.33      0.33      0.33      9842
weighted avg       0.33      0.33      0.33      9842
```

In [41]:

```
# one sideed
pipe=Pipeline([("vectorize",CountVectorizer(binary=False,ngram_range=(1,1))
pipe.fit(df_train["sentence1"],df_train["gold_label"])
os_s1_res=pipe.predict(df_dev["sentence1"])
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

In [42]:

```
print(classification_report(df_dev["gold_label"],os_s1_res))
print(confusion_matrix(df_dev["gold_label"],os_s1_res))
```

```
              precision    recall  f1-score   support

contradiction       0.34      0.27      0.30      3278
   entailment       0.34      0.40      0.37      3329
      neutral       0.33      0.34      0.34      3235

     accuracy                           0.34      9842
    macro avg       0.34      0.34      0.34      9842
 weighted avg       0.34      0.34      0.34      9842

[[ 884 1302 1092]
 [ 874 1340 1115]
 [ 856 1268 1111]]
```

In [43]:

```
pipe.fit(df_train["sentence2"],df_train["gold_label"])
os_s2_res=pipe.predict(df_dev["sentence2"])
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

In [44]:

```
print(confusion_matrix(df_dev["gold_label"],os_s2_res))
print(classification_report(df_dev["gold_label"],os_s2_res))
```

```
[[2077  654  547]
 [ 554 2320  455]
 [ 603  703 1929]]
              precision    recall  f1-score   support

contradiction       0.64      0.63      0.64      3278
   entailment       0.63      0.70      0.66      3329
      neutral       0.66      0.60      0.63      3235

     accuracy                           0.64      9842
    macro avg       0.64      0.64      0.64      9842
 weighted avg       0.64      0.64      0.64      9842
```

In [45]:

```
# cross_unigram
pipe=Pipeline([("transform",CrossUnigramsTransformer()),("vectorize",CountVe
pipe.fit(df_train.drop(["gold_label"],axis=1),df_train["gold_label"])

cu_res=pipe.predict(df_dev.drop(["gold_label"],axis=1))
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\linea
r_model\_logistic.py:764: ConvergenceWarning: lbfgs failed to converge (stat
us=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regre
ssion
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

In [46]:

```
print(confusion_matrix(df_dev["gold_label"],cu_res))
print(classification_report(df_dev["gold_label"],cu_res))
```

```
[[2066  694  518]
 [ 679 2207  443]
 [ 705  762 1768]]
              precision    recall  f1-score   support

contradiction       0.60      0.63      0.61      3278
   entailment       0.60      0.66      0.63      3329
      neutral       0.65      0.55      0.59      3235

     accuracy                           0.61      9842
    macro avg       0.62      0.61      0.61      9842
 weighted avg       0.62      0.61      0.61      9842
```

In [47]:

```
# MLP
pipe=Pipeline([("vectorize",PairedSentenceVectorizer()),("MLP",MLPClassifier
pipe.fit(df_train.drop(["gold_label"],axis=1),df_train["gold_label"],)
MLP_res=pipe.predict(df_dev.drop(["gold_label"],axis=1))
print(confusion_matrix(df_dev["gold_label"],MLP_res))
```

```
C:\Users\Administrator\anaconda3\envs\kaggle\lib\site-packages\sklearn\neura
l_network\_multilayer_perceptron.py:585: ConvergenceWarning: Stochastic Opti
mizer: Maximum iterations (8) reached and the optimization hasn't converged
```

```
yet.
[[2437  347  494]
 [ 157 2697  475]
 [ 373  455 2407]]
```

In [48]:
```
print(classification_report(df_dev["gold_label"],MLP_res))
```

```
              precision    recall  f1-score   support

contradiction       0.82      0.74      0.78      3278
   entailment       0.77      0.81      0.79      3329
      neutral       0.71      0.74      0.73      3235

     accuracy                           0.77      9842
    macro avg       0.77      0.77      0.77      9842
 weighted avg       0.77      0.77      0.77      9842
```

TODO: Insert your discussion of the experimental results here

The results on full training data have the same pattern comparing to partial training data. Random baseline had the worst result then onesided baseline, cross-unigram performs slightly worse than onesided baseline and MLP has the best accuracy rate among them. With full training data, our classifier was fully train with enough information, so it will have a better result on validation with same validation data.

Please read the section ‘General information’ on the ‘Labs’ page of the course website before submitting this notebook!