IN4080 – Natural Language Processing

This assignment has two parts:

- Part A. Sequence labeling
- Part B. Word embeddings

Part A

In this part we will experiment with sequence classification and tagging. We will combine some of the tools for tagging from NLTK with scikit-learn to build various taggers.We will start with simple examples from NLTK where the tagger only considers the token to be tagged—not its context— and work towards more advanced logistic regression taggers (also called maximum entropy taggers). Finally, we will compare to some tagging algorithms installed in NLTK.

In [1]:
```python
import re
import pprint
import nltk
from nltk.corpus import brown
tagged_sents = brown.tagged_sents(categories='news')
size = int(len(tagged_sents) * 0.1)
train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]
```

In [2]:
```python
def pos_features(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features

class ConsecutivePosTagger(nltk.TaggerI):
    def __init__(self, train_sents, features=pos_features):
        self.features = features
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)
    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = self.features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
```

In [3]:
```python
tagger = ConsecutivePosTagger(train_sents)
print(round(tagger.evaluate(test_sents), 4))
```

```
0.7915
```

1) Tag set and baseline

Part a: Tag set and experimental set-up

In [4]:
```python
def split_data(tagged_sents_uni):
    size = len(tagged_sents_uni)
    slice_ind = round(size*10/100)
    news_test = tagged_sents_uni[:slice_ind]
    news_dev_test = tagged_sents_uni[slice_ind:slice_ind*2]
    news_train = tagged_sents_uni[slice_ind*2:]

    return news_test, news_dev_test,news_train
```

In [5]:
```python
tagged_sents_uni = brown.tagged_sents(categories='news',tagset = 'universal')
news_test, news_dev_test,news_train = split_data(tagged_sents_uni)
```

In [6]:
```python
tagger_a = ConsecutivePosTagger(news_train)
print(round(tagger_a.evaluate(news_dev_test), 4))
```

```
0.8689
```

We got higher accuracy.

Part b: One of the first things we should do in an experiment like this, is to establish a reasonable baseline. A reasonable baseline here is the Most Frequent Class baseline. Each word which is seen during training should get its most frequent tag from the training. For words not seen during training, we simply use the most frequent overall tag. For this task, we can use a simple UnigramTagger:

In [ ]:
```python
baseline_tagger = nltk.UnigramTagger(news_train)
round(baseline_tagger.evaluate(news_dev_test), 4)
```

2) scikit-learn and tuning

Our goal will be to improve the tagger compared to the simple suffix-based tagger. For the further experiments, we move to scikit-learn which yields more options for considering various alternatives. We have reimplemented the ConsecutivePosTagger to use scikit-learn classifiers below. We have made the classifier a parameter so that it can easily be exchanged. We start with the BernoulliNBclassifier which should correspond to the way it is done in NLTK.

In [7]:
```python
import numpy as np
import sklearn

from sklearn.naive_bayes import BernoulliNB
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction import DictVectorizer


class ScikitConsecutivePosTagger(nltk.TaggerI):

    def __init__(self, train_sents, features=pos_features, clf = BernoulliNB()):
        # Using pos_features as default.
        self.features = features
        train_features = []
        train_labels = []
        for tagged_sent in train_sents:
            history = []
            untagged_sent = nltk.tag.untag(tagged_sent)
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = features(untagged_sent, i, history)
                train_features.append(featureset)
                train_labels.append(tag)
                history.append(tag)
        v = DictVectorizer()
        X_train = v.fit_transform(train_features)
        y_train = np.array(train_labels)
        clf.fit(X_train, y_train)
        self.classifier = clf
        self.dict = v

    def tag(self, sentence):
        test_features = []
        history = []
        for i, word in enumerate(sentence):
            featureset = self.features(sentence, i, history)
            test_features.append(featureset)
        X_test = self.dict.transform(test_features)
        tags = self.classifier.predict(X_test)
        return zip(sentence, tags)
```

Part a) Training the ScikitConsecutivePosTagger with *news_train* set and test on the *news_dev_test* set with the *pos_features*.

In [8]:
```python
tagger_scikit = ScikitConsecutivePosTagger(news_train)
print(round(tagger_scikit.evaluate(news_dev_test), 4))
```

```
0.857
```

We can see that, by using the same data and same features we get a bit inferior results.

Part b) One explanation could be that the smoothing is too strong. *BernoulliNB()* from scikit-learn uses Laplace smoothing as default ("add-one"). The smoothing is generalized to Lidstone smoothing which is expressed by the alpha parameter to *BernoulliNB(alpha=...)*. Therefore, we will tune the alpha parameter to find the most optimal one.

In [9]:
```python
def tunning_bernoulli(pos_features):
    alphas = [1, 0.5, 0.1, 0.01, 0.001, 0.0001]
    accuracies = []
    for alpha in alphas:
        tagger_sci = ScikitConsecutivePosTagger(news_train,features = pos_features ,clf = BernoulliNB(alpha=alp
        accuracies.append(round(tagger_sci.evaluate(news_dev_test), 4))

    return alphas,accuracies

def visualize_results(alphas, accuracies):
    acc_alphas = {'alpha':alphas,'Accuracies':accuracies}
    import pandas as pd
    df = pd.DataFrame(acc_alphas)
    print(df)

    best_acc = max(accuracies)
    best_ind = accuracies.index(max(accuracies))
    best_alpha = alphas[best_ind]
    print("")
    print(f'Best alpha: {best_alpha} - accuracy: {best_acc}')
```

In [10]:
```python
alphas,accuracies = tunning_bernoulli(pos_features)
```

In [11]:
```python
visualize_results(alphas, accuracies)
```

```
    alpha  Accuracies
0  1.0000      0.8570
1  0.5000      0.8749
2  0.1000      0.8695
3  0.0100      0.8683
4  0.0010      0.8651
5  0.0001      0.8631

Best alpha: 0.5 - accuracy: 0.8749
```

We can see that we get a little bit better result with Scikits BernoulliNB with the best alpha.

Part c) To improve the results, we may change the feature selector or the machine learner. We start with a simple improvement of the feature selector. The NLTK selector considers the previous word, but not the word itself. Intuitively, the word itself should be a stronger feature. By extending the NLTK feature selector with a feature for the token to be tagged, we try to find the best results.

In [12]:
```python
def pos_features_tagged(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]

    #same structure, but included the token to be tagged.
    features['tagged_word'] = sentence[i]

    return features
```

In [13]:
```python
alphas_tag,accuracies_tag = tunning_bernoulli(pos_features_tagged)
visualize_results(alphas_tag, accuracies_tag)
```

```
    alpha  Accuracies
0  1.0000      0.8874
1  0.5000      0.9166
2  0.1000      0.9244
3  0.0100      0.9303
4  0.0010      0.9330
5  0.0001      0.9340

Best alpha: 0.0001 - accuracy: 0.934
```

3) Logistic regression

Part a) We proceed with the best feature selector from the last exercise. We will study the effect of the learner.

In [14]:
```python
from sklearn.linear_model import LogisticRegression

#increased the max_iter from default 100 to 500 in order to make it converge:
logClf = LogisticRegression(max_iter = 500)
```

In [15]:
```python
tagger_log = ScikitConsecutivePosTagger(news_train,features = pos_features ,clf = logClf)
acc_log = (round(tagger_log.evaluate(news_dev_test), 4))
print(f'Logistic accuracy = {acc_log}')
```

```
Logistic accuracy = 0.8996
```

The *Logistic Regression* classifier is better than all of the *BernoulliNB* methods without the token to be tagged.

Part b) Similarly to the Naive Bayes classifier, we will study the effect of smoothing. Smoothing for LogisticRegression is done by regularization. In scikit-learn, regularization is expressed by the parameter C. A smaller C means a heavier smoothing (C is the inverse of the parameter α in the lectures). We will tune the C parameter in order to find the most optimal model.

In [16]:
```python
def tunning_logistic(pos_features):
    C_values = [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
    accuracies = []
    for C in C_values:
        print(f"Running: LogisticRegression(C = {C})")
        logClf = LogisticRegression(C=C,max_iter = 10000)
        tagger_log = ScikitConsecutivePosTagger(news_train,features = pos_features ,clf = logClf)
        accuracies.append(round(tagger_log.evaluate(news_dev_test), 4))

    return C_values,accuracies
```

In [17]:
```python
C_values,accuracies_log = tunning_logistic(pos_features)
```

```
Running: LogisticRegression(C = 0.01)
Running: LogisticRegression(C = 0.1)
Running: LogisticRegression(C = 1.0)
Running: LogisticRegression(C = 10.0)
Running: LogisticRegression(C = 100.0)
Running: LogisticRegression(C = 1000.0)
```

In [18]:
```python
visualize_results(C_values, accuracies_log)
```

```
     alpha  Accuracies
0     0.01      0.8321
1     0.10      0.8827
2     1.00      0.8996
3    10.00      0.8998
4   100.00      0.8949
5  1000.00      0.8896

Best alpha: 10.0 - accuracy: 0.8998
```

4) Features

Part a) We will now stick to the LogisticRegression() with the optimal C from the last point and see whether we are able to improve the results further by extending the feature extractor with more features. First, try adding a feature for the next word in the sentence, and then train and test.

In [19]:
```python
def pos_features_extended(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]

    #next word in the secquence:
    if i == len(sentence) - 1:
        features['next-word'] = sentence[i]
    else:
        features['next-word'] = sentence[i+1]

    return features
```

In [20]:
```python
def find_accuracy(pos_features,news_train,news_dev_test):
    best_ind = accuracies_log.index(max(accuracies_log))
    optimal_C = C_values[best_ind]

    clf = LogisticRegression(C=optimal_C,solver= 'liblinear')
    tagger = ScikitConsecutivePosTagger(news_train,features = pos_features ,clf = clf)
    acc = (round(tagger.evaluate(news_dev_test), 4))
    return acc
```

In [21]:
```python
acc_opt_log = find_accuracy(pos_features_extended,news_train,news_dev_test)
print(f'Logistic regression with optimal C: {acc_opt_log}')
```

```
Logistic regression with optimal C: 0.9231
```

Part b) We will continue to add more features to get an even better tagger.

In [22]:
```python
def pos_features_decapilized(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]

    #next word in the secquence:
    if i == len(sentence) - 1:
        features['next-word'] = sentence[i]
    else:
        features['next-word'] = sentence[i+1]
    features['current-word'] = sentence[i]


    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

    s = sentence[i]

    if s.isupper():
        s = s.lower()
    elif s.isdigit():
        features['type'] = 'digit'
    elif s in punctuation:
        features['type'] = 'punctuation'
    else:
        features['type'] = 'other'

    return features
```

In [23]:
```python
acc_extended = find_accuracy(pos_features_decapilized,news_train,news_dev_test)
print(f'Logistic regression with optimal C: {acc_extended}')
```

```
Logistic regression with optimal C: 0.9669
```

By adding the current word, we get very much more improvement.

5) Larger corpus and evaluation

Part a) We will now test our best tagger so far on the news_test set.

In [24]:
```python
acc_test_data = find_accuracy(pos_features_decapilized,news_train,news_test)
print(f'Logistic regression - accuracy = {acc_test_data}')
```

```
Logistic regression - accuracy = 0.9678
```

Part b) Now,we will use nearly the whole Brown corpus. But we will take away two categories for later evaluation: *adventure* and *hobbies*. We will also initially stay clear of *news* to be sure not to mix training and test data.

In [25]:
```python
categories = brown.categories()
categories.remove('news')
categories.remove('adventure')
categories.remove('hobbies')
tagged_sents = brown.tagged_sents(categories=categories)
brown_data = brown.tagged_sents(categories = categories , tagset = 'unvisersal')

rest_test, rest_dev_test,rest_train = split_data(brown_data)
```

In [26]:
```python
#merging the datasets

train = rest_train + news_train
test = rest_test + news_test
dev_test = rest_dev_test + news_dev_test
```

In [28]:
```python
baseline_tagger = nltk.UnigramTagger(train)
round(baseline_tagger.evaluate(test), 4)
```

Out[28]:
```
0.8451
```

Part c) We can then build our tagger for this larger domain. By using the best setting, we will try to find the accuracy for this dataset.

In [30]:
```python
#acc_large = find_accuracy(pos_features_decapilized,train,test)
best_ind = accuracies_log.index(max(accuracies_log))
optimal_C = C_values[best_ind]

optimal_clf = LogisticRegression(C=optimal_C,solver= 'liblinear',max_iter = 1000)
tagger_domain = ScikitConsecutivePosTagger(train,features = pos_features_decapilized ,clf = optimal_clf)
acc_domain = (round(tagger_domain.evaluate(test), 4))
```

In [31]:
```python
print(f'The accuracy for the tagger for whole domain = {acc_domain}')
```

```
The accuracy for the tagger for whole domain = 0.8732
```

Part d) Now, testing the big tagger on *adventure* and *hobbies* categories of Brown corpus.

In [32]:
```python
adventures_sents = brown.tagged_sents(categories = 'adventure' , tagset = 'unvisersal')
hobbies_sents = brown.tagged_sents(categories = 'hobbies' , tagset = 'unvisersal')
```

In [33]:
```python
acc_adventure = (round(tagger_domain.evaluate(adventures_sents), 4))
acc_hobbies = (round(tagger_domain.evaluate(hobbies_sents), 4))
```

In [34]:
```python
print(f'Accuracy: {acc_adventure} - adventures')
print(f'Accuracy: {acc_hobbies} - hobbies')
```

```
Accuracy: 0.9889 - adventures
Accuracy: 0.9776 - hobbies
```

We can see here that the accuracy for the *adventures* were a bit better than for the *hobbies*. One explanation for this could be that *adventures* text is written in a formel like the trainingset, while the *hobbies* contains words and phrases in subjective form.

6) Comparing to other taggers

Part a) NLTK comes with an HMM-tagger which we may train and test on our own corpus. It can be trained and testet by

In [35]:
```python
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(news_train)
news_hmm_acc = round(news_hmm_tagger.evaluate(news_test), 4)
print(f"The news HMM tagger accuracy: {news_hmm_acc}")
```

```
The news HMM tagger accuracy: 0.8995
```

Training and testing on the whole data:

In [36]:
```python
big_hmm_tagger = nltk.HiddenMarkovModelTagger.train(train)
big_hmm_acc = round(big_hmm_tagger.evaluate(test), 4)
print(f"The HMM tagger accuracy: {big_hmm_acc}")
```

```
The HMM tagger accuracy: 0.6738
```

This method of tagging has better speed for training og evaluating, however the accuracy is not quite good.

Part b) NLTK also comes with an averaged perceptron tagger which we may train and test. It is currently considered the best tagger included with NLTK. It can be trained as follows:

In [37]:
```python
def run_per_tagger(train,test,name):
    per_tagger = nltk.PerceptronTagger(load=False)
    per_tagger.train(train)
    per_acc = round(per_tagger.evaluate(test), 4)

    print(f'Perceptron tagger accuracy: {per_acc} - {name}')
```

In [38]:
```python
run_per_tagger(news_train,news_test,'news_data')
run_per_tagger(news_train,news_test,'all_data')
```

```
Perceptron tagger accuracy: 0.9656 - news_data
Perceptron tagger accuracy: 0.9658 - all_data
```

This is definitely the tagger in this assignment, both in terms of speed and accuracy. It got much better results for *train* data than the best tagger above, but did the computing in much less time. However, it did not as good accuracy as the best model for the *news_data*.

Part B

In this part we will use the gensim package to familiarize ourselves with word embeddings and **word2vec**.

In [39]:
```python
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

1) Basics

a) The amount of different words in the model:

In [40]:
```python
total_words = len(wv)
print(f'Total words in the model: {total_words}')
```

```
Total words in the model: 3000000
```

In [41]:
```python
try:
    vec_cameroon = wv['cameroon']
except KeyError:
    print("The word 'cameroon' does not appear in this model")
```

```
The word 'cameroon' does not appear in this model
```

b)) Implementing a function for calculating the norm (the length) of an (embedding) vector, and a function for calculating the cosine between two vectors.

In [42]:
```python
import numpy as np
def norm(vector):
    return np.linalg.norm(vector)

def similarity(vector1, vector2):
    cosine = np.dot(vector1,vector2)/(norm(vector1) * norm(vector2))
    return cosine
```

c) Comparing the functions with:

In [43]:
```python
print(wv.similarity('king','queen'))
print(similarity(wv['king'],wv['queen']))
```

```
0.6510957
0.6510957
```

2) Built in functions

Several built-in functions let you inspect semantic properties of the embeddings. The *most_similar* lets you find the nearest neighbor to one or more words.

In [44]:
```python
print(wv.most_similar('car', topn=5))
print(wv.most_similar(positive=['car', 'minivan'], topn=5))
```

```
[('vehicle', 0.7821096181869507), ('cars', 0.7423830032348633), ('SUV', 0.7160962224006653), ('minivan', 0.6907
036304473877), ('truck', 0.6735789775848389)]
[('SUV', 0.8532191514968872), ('vehicle', 0.8175783753395081), ('pickup_truck', 0.7763689160346985), ('Jeep',
0.7567334175109863), ('Ford_Explorer', 0.7565719485282898)]
```

It is also the tool for testing analogies, e.g. "Norway is to Oslo as Sweden is to ..." as

In [45]:
```python
print(wv.most_similar(positive=['Oslo', 'Sweden'],negative = ['Norway'], topn=5))
```

```
[('Stockholm', 0.7886312007904053), ('Helsinki', 0.648445725440979), ('Stockholm_Sweden', 0.6368898153305054),
('Malmö', 0.6361426711082458), ('Oslo_Norway', 0.6240758299827576)]
```

a) Trying these analogy tests:

" king is to man as queen is to ..."

" king is to queen as man is to ..."

"cat is to kitten as dog is to ..."

In [46]:
```python
print(wv.most_similar(positive=['man', 'queen'],negative = ['king'], topn=5))
```

```
[('woman', 0.7609436511993408), ('girl', 0.6139993667602539), ('teenage_girl', 0.6040961742401123), ('teenage
r', 0.5825759172439575), ('lady', 0.5752554535865784)]
```

In [47]:
```python
print(wv.most_similar(positive=['queen', 'man'],negative = ['king'], topn=5))
```

```
[('woman', 0.7609436511993408), ('girl', 0.6139993667602539), ('teenage_girl', 0.6040961742401123), ('teenage
r', 0.5825759172439575), ('lady', 0.5752554535865784)]
```

In [48]:
```python
print(wv.most_similar(positive=['kitten', 'dog'],negative = ['cat'], topn=5))
```

```
[('puppy', 0.7699725031852722), ('pup', 0.6861710548400879), ('pit_bull', 0.6776558756828308), ('dogs', 0.67709
86318588257), ('Rottweiler', 0.6646621823310852)]
```

b) To understand the method better, we can try to follow the recipe more directly.

In [49]:
```python
a = wv['king'] + wv['woman'] - wv['man']
strings = ['king','queen','man','women']

similarity_dict = {s:similarity(a,wv[s]) for s in strings }
similarity_wv = wv.similar_by_vector(a)
```

In [50]:
```python
print(similarity_wv)
```

```
[('king', 0.8449392318725586), ('queen', 0.7300518155097961), ('monarch', 0.645466148853302), ('princess', 0.61
56251430511475), ('crown_prince', 0.5818676948547363), ('prince', 0.5777117609977722), ('kings', 0.561366438865
6616), ('sultan', 0.5376776456832886), ('Queen_Consort', 0.5344247221946716), ('queens', 0.5289887189865112)]
```

In [51]:
```python
print(similarity_dict)
```

```
{'king': 0.84493923, 'queen': 0.73005176, 'man': 0.121606365, 'women': 0.25710744}
```

This illustrates how the *most_similar* works. We tried for *a* here, meaning that we wanted to know with word most near the 'queen'. By using the cosine between the vectors, we can see above that the 'women' with '0.25710' is most near.

c) doesnt_match:

In [52]:
```python
print(wv.doesnt_match(['Norway', 'Denmark', 'Finland','Sweden', 'Spain', 'Stockholm']))
```

```
Spain
```

In [53]:
```python
print(wv.doesnt_match(['Oslo', 'Bergen', 'Trondheim', 'Ålesund', 'Somalia']))
```

```
Somalia
```

In [54]:
```python
print(wv.doesnt_match(['runing','swimming','sprinting', 'sleeping','bodybuilding']))
```

```
sleeping
```

In [55]:
```python
#It can not classify the African countries correctly.
print(wv.doesnt_match(['Kenya','Somalia','Libya','Egypt', 'Indonesia']))
```

```
Libya
```

In [56]:
```python
print(wv.doesnt_match(['book','read','potato','school','coffee']))
#potato most likely does not match here
```

```
school
```

3) Training a toy model

In [57]:
```python
from gensim.test.utils import datapath
from gensim import utils
import gensim.models
from nltk.corpus import brown
```

In [58]:
```python
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
sentences = brown.sents()
model = gensim.models.Word2Vec(sentences)
```

The Brown corpus is relatively smaller corpus compared to Google News corpus. Brown corpus contains around 1 million words while Google News contains approximately 100 billion words.

b) Comparing Brown model to the 'word2vec-google-news-300'

In [59]:
```python
google_car = wv.most_similar('car', topn=10)
google_queen = wv.most_similar('queen', topn=10)

brown_car = model.wv.most_similar('car', topn=10)
brown_queen = model.wv.most_similar('queen', topn=10)
```

In [60]:
```python
print(brown_car)
print(google_car)
```

```
[('house', 0.9468015432357788), ('hall', 0.904809296131134), ('room', 0.9046733379364014), ('corner', 0.9016127
586364746), ('town', 0.9007890820503235), ('road', 0.894270122051239), ('desk', 0.8923514485359192), ('bed', 0.
8916454911231995), ('jig', 0.8833274841308594), ('ball', 0.8798673748970032)]
[('vehicle', 0.7821096181869507), ('cars', 0.7423830032348633), ('SUV', 0.7160962224006653), ('minivan', 0.6907
036304473877), ('truck', 0.6735789775848389), ('Car', 0.6677608489990234), ('Ford_Focus', 0.6673202514648438),
('Honda_Civic', 0.6626849174499512), ('Jeep', 0.651133120059967), ('pickup_truck', 0.6441437602043152)]
```

In [61]:
```python
print(brown_queen)
print(google_queen)
```

```
[('calf', 0.9515677690505981), ('gown', 0.9503137469291687), ('Sloan', 0.9497048258781433), ('governor', 0.9474
377632141113), ('gentle', 0.947225034236908), ('shock', 0.9448780417442322), ('nervous', 0.9446044564247131),
('surgeon', 0.9406843781471252), ('blonde', 0.9385437965393066), ('cigarette', 0.9380086660385132)]
[('queens', 0.7399442791938782), ('princess', 0.7070531249046326), ('king', 0.6510956883430481), ('monarch', 0.
6383602023124695), ('very_pampered_McElhatton', 0.6357026696205139), ('Queen', 0.6163408160209656), ('NYC_anglo
philes_aflutter', 0.6060680150985718), ('Queen_Consort', 0.5923796892166138), ('princesses', 0.590807497501373
3), ('royal', 0.5637185573577881)]
```

c)

In [62]:
```python
print(model.wv.most_similar(positive=['man', 'queen'],negative = ['king'], topn=5))
print(model.wv.most_similar(positive=['kitten', 'dog'],negative = ['cat'], topn=5))
print(model.wv.most_similar(positive=['queen', 'man'],negative = ['king'], topn=5))
```

```
[('boy', 0.8090471029281616), ('girl', 0.8031275868415833), ('woman', 0.7906173467636108), ('himself', 0.720608
4132194519), ('young', 0.7140515446662903)]
[('greeting', 0.8697917461395264), ('enrich', 0.8523256778717041), ('choose', 0.849356472492218), ('follow', 0.
8472919464111328), ('begin', 0.8413951992988586)]
[('boy', 0.8090471029281616), ('girl', 0.8031275868415833), ('woman', 0.7906173467636108), ('himself', 0.720608
4132194519), ('young', 0.7140515446662903)]
```

4) Evaluation

Gensim comes with several methods for evaluation together with standard datasets for the tests. Testsets can be found by the tha datapath command, e.g.

In [63]:
```python
path=datapath('questions-words.txt')
```

One test we may use is to see how well the model perform on the Google analogy test datset. This can be run by

In [64]:
```python
model_evaluation = wv.evaluate_word_analogies(path)
```

In [75]:
```python
model_evaluation[0]
```

Out[75]:
```
0.7401448525607863
```

5) Application

We will try a simple example of applying word embeddings to an NLP task. We consider text classification. We will use the same movie dataset from NLTK as we used in Mandatory assignment 1B, with the same split as we used there. Thereby, we may compare the results with the results from Mandatory 1. We will consider a document as a bag of words. The word order and sentence structure will be ignored. Each word can be represented by its embedding. But how should a document be represented? The easiest is to use the "semantic fingerprint", which means representing the document by the average vector of its words.

In [66]:
```python
import random
from nltk.corpus import movie_reviews
import numpy as np
import scipy as sp
import sklearn
from sklearn.linear_model import LogisticRegression
```

In [67]:
```python
raw_movie_docs = [(movie_reviews.raw(fileid), category) for category in movie_reviews.categories()
                  for fileid in movie_reviews.fileids(category)]

random.seed(2920)
random.shuffle(raw_movie_docs)

##tokenizing the text and finding every words vector
def tokenize_and_embedding(text_data):
    data = []

    for i in range(len(text_data)):
        embeddings = []
        token_data = nltk.word_tokenize(text_data[i][0])

        for w in token_data:
            if w in wv:
                embeddings.append(wv[w])
        data.append([embeddings,text_data[i][1]])
    return data

raw_movie_docs = tokenize_and_embedding(raw_movie_docs)
```

In [68]:
```python
movie_test = raw_movie_docs[:200]
movie_dev = raw_movie_docs[200:]
train_data = movie_dev[:1600]
dev_test_data = movie_dev[1600:]
```

In [69]:
```python
def split_target_text(text_data):
    target = []
    texts = []
    for doc in text_data:
        texts.append(doc[0])
        target.append(doc[1])

    return target,texts
```

In [70]:
```python
train_target, train_texts = split_target_text(train_data)
dev_test_target, dev_test_texts = split_target_text(dev_test_data)
```

In [71]:
```python
def find_mean(train_texts):
    list_of_means = []
    for i in range(len(train_texts)):
        text = train_texts[i]
        text = np.array(text)
        mean_of_text = []
        for j in range(len(text[0])):
            mean_of_text.append(np.mean(text[:,j]))
        list_of_means.append(mean_of_text)
    return list_of_means
```

In [72]:
```python
train_texts = find_mean(train_texts)
dev_test_texts = find_mean(dev_test_texts)
```

In [73]:
```python
def tunning_logistic():
    print("Running logistic regression classifier: \n")
    C_values = [0.01, 0.1, 1.0, 10.0, 100.0,500.0, 1000.0,2000.0]
    for C in C_values:
        log_reg = LogisticRegression(solver='liblinear',C=C)
        log_reg.fit(train_texts,train_target)
        acc = log_reg.score(dev_test_texts,dev_test_target)
        print(f'C = {C:7.2f} - accuracy = {acc:4.4f}')
```

In [74]:
```python
tunning_logistic()
```

```
Running logistic regression classifier: 

C =    0.01 - accuracy = 0.6950
C =    0.10 - accuracy = 0.7150
C =    1.00 - accuracy = 0.7950
C =   10.00 - accuracy = 0.8250
C =  100.00 - accuracy = 0.8500
C =  500.00 - accuracy = 0.8300
C = 1000.00 - accuracy = 0.8250
C = 2000.00 - accuracy = 0.8250
```

We can see here that the best accuracy was achieved for C = 100. However, this accuracy is not as good as the accuracy found in Mandatory 1.