in4080_2020_mandatory_2

October 16, 2020

1 IN4080, 2020, Mandatory assignment 2, part A

Markus Heiervang - markuhei

1.0.1 About the assignment

Your answer should be delivered in devilry no later than Friday, 9 October at 23:59
Mandatory assignment 2 consists of two parts

- Part A on tagging and sequence classification (=this file)
- Part B on word embeddings (separate document)

You should answer both parts. It is possible to get 65 points part A, 35 points on part B, 100 points altogether. You are required to get at least 60 points to pass. It is more important that you try to answer each question than that you get it correct.

1.0.2 General requirements:

- We assume that you have read and are familiar with IFI's requirements and guidelines for mandatory assignments
  - https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-mandatory.html
  - https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-guidelines.html
- This is an individual assignment. You should not deliver joint submissions.
- You may redeliver in Devilry before the deadline, but include all files in the last delivery. Only the last delivery will be read!
- If you deliver more than one file, put them into a zip-archive.
- Name your submission your_username_in4080_mandatory_2

The delivery can take one of two forms:

- Alternative A:
  - Deliver the code files.
  - In addition, deliver a separate pdf file containing results from the runs together with answers to the text questions.

- Alternative B:
  - A jupyter notebook containing code, answers to the text questions in markup and optionally results from the runs.
  - In addition, a pdf-version of the notebook where (in addition) all the results of the runs are included.

Whether you use the first or second alternative, make sure that the code runs at the IFI machines after

- export PATH=/opt/ifi/anaconda3/bin/:$PATH

or on your own machine in the environment we provided, see - https://www.uio.no/studier/emner/matnat/ifi/IN4080/h20/lab-setup/.

If you use any additional packages, they should be included in your delivery.

1.0.3 Goals of 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 this set you will get more experience with

- baseline for a tagger and more generally for classification
- how different tag sets may result in different accuracies
- feature selection
- the effect of the machine learner
- smoothing
- evaluation
- in-domain and out-of-domain evaluation

To get a good tagger, you need a reasonably sized tagged training corpus. Ideally, we would have used the complete Brown corpus in this exercise, but it turns out that some of the experiments we will run, will be time consuming. Hence, we will follow the NLTK book and use only the News section. Since this is a rather homogeneous domain, and we also pick our test data from the same domain, we can still get decent results.

Towards the end of the exercise set, we will see what happens if we take our best settings from the News section to a bigger domain.

Beware that even with this reduced corpus, some of the experiments will take several minutes. And when we build the full tagger in exercise 5, an experiment may take half an hour. So make sure you start the work early enough. (You might do other things while the experiments are running.)

1.0.4 Replicating NLTK Ch. 6

We jump into the NLTK book, chapter 6, the sections 6.1.5 Exploiting context and 6.1.6 Sequence classification. You are advised to read them before you start.

We start by importing NLTK and the tagged sentences from the news-section from Brown, similarly to the NLTK book.

Then we split the set of sentences into a train set and a test set.

```
In [2]: import re
        import pprint
        import nltk
        from nltk.corpus import brown
        import time
        import random
        import gensim
        import gensim.downloader as api

        SEED = 42

        tagged_sents = brown.tagged_sents(categories='news')
        size = int(len(tagged_sents) * 0.1)
        train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]
```

Like NLTK, our tagger will have two parts, a feature extractor, here called **pos_features**, and a general class for building taggers, **ConsecutivePosTagger**.

We have made a few adjustments to the NLTK setup. We are using the *pos_features* from section 6.1.5 together with the *ConsecutivePosTagger* from section 6.1.6. The *pos_features* in section 1.5 does not consider history, but to get a format that works together with *ConsecutivePosTagger*, we have included an argument for history in *pos_features*, which is not used initially. (It get used by the *pos_features* in section 6.1.6 of the NLTK book, and you may choos to use it later in this set).

Secondly, we have made the *feature_extractor* a parameter to *ConsecutivePosTagger*, so that it can easily be replaced by other feature extractors while keeping *ConsecutivePosTagger*.

```
In [8]: 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

In [9]: 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)
```

Following the NLTK bok, we train and test a classifier.

```
In [180]: tagger = ConsecutivePosTagger(train_sents)
          print(round(tagger.evaluate(test_sents), 4))
```

```
0.7915
```

This should give results comparable to the NLTK book.

1.1 Ex 1: Tag set and baseline (10 points)

1.1.1 Part a. Tag set and experimental set-up

We will simplify and use the universal pos tagset in this exercise. One main reason is that it makes the experiments run faster.

We will be a little more cautious than the NLTK-book, when it comes to training and test sets. We will split the News-section into three sets

- 10% for final testing which we tuck aside for now, call it *news_test*
- 10% for development testing, call it *news_dev_test*
- 80% for training, call it *news_train*
- Make the data sets, and repeat the training and evaluation with *news_train* and *news_dev_test*.
- Please use 4 counting decimal places and stick to that throughout the exercise set.

How is the result compared to using the full brown tagset? Why do you think one of the tagsets yields higher scores than the other one?

```
In [10]: tagged_sents = brown.tagged_sents(categories='news', tagset='universal')

         def train_dev_test_split(original_data, ratios=(0.8, 0.1), seed=42):
             assert sum(ratios)+ratios[1] == 1.0
             random.seed(SEED)
             data = list(original_data)
             random.shuffle(data)
```

```
            N = len(data)
            a, b = int(N*ratios[0]), int(N*ratios[1])
            train = data[:a]
            dev = data[a:-b]
            test = data[-b:]
            return train, dev, test

        news_train, news_dev, news_test = train_dev_test_split(tagged_sents)
```

```
In [184]: tagger = ConsecutivePosTagger(news_train)
          print(round(tagger.evaluate(news_dev), 4))
```

```
0.8669
```

The results were better. The universal tagset contains fewer tags, meaning that the classification becomes more narrow. In turn, this means we have more datasamples per class.

1.1.2 Part b. Baseline

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.

With news_train as training set and news_dev_set as valuation set, what is the accuracy of this baseline?

Does the tagger from part (a) using the features from the NLTK book beat the baseline?

```
In [11]: class BaselineTagger(nltk.TaggerI):
             def __init__(self, train_set):
                 cfd = nltk.ConditionalFreqDist((w.lower(), t) for doc in news_train for w, t
                 self.mc_labels = {term: dist.max() for term, dist in cfd.items()}
                 self.most_common_label = nltk.FreqDist(self.mc_labels).most_common()

             def tag(self, sentence):
                 return ((term, self.mc_labels.get(term.lower(), self.most_common_label)) for

         blt = BaselineTagger(news_train)
         round(blt.evaluate(news_dev), 4)
```

```
Out[11]: 0.8846
```

No.

1.2 Ex2: scikit-learn and tuning (10 points)

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 BernoulliNB-classifier which should correspond to the way it is done in NLTK.

```
In [12]: 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)
```

1.2.1 Part a.

Train the ScikitConsecutivePosTagger on the *news_train* set and test on the *news_dev_test* set with the *pos_features*. Do you get the same result as with the same data and features and the NLTK code in exercise 1a?

```
In [ ]: tagger = ScikitConsecutivePosTagger(news_train)
        round(tagger.evaluate(news_dev), 4)

Out[ ]: 0.8474
```

No

1.2.2 Part b.

I get inferior results compared to using the NLTK set-up with the same feature extractors. The only explanation I could find is 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, try again with alpha in [1, 0.5, 0.1, 0.01, 0.001, 0.0001]. What do you find to be the best value for alpha?

With the best choice of alpha, do you get the same results as with the NLTK code in exercise 1a, worse results or better results?

```
In [226]: alphas = [1, 0.5, 0.1, 0.01, 0.001, 0.0001]

          for a in alphas:
              tagger = ScikitConsecutivePosTagger(news_train, clf=BernoulliNB(alpha=a))
              print(round(tagger.evaluate(news_dev), 4))
```

```
0.8474
0.8693
0.8714
0.8728
0.8715
0.8695
```

Most of the results are better

1.2.3 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. Extend the NLTK feature selector with a feature for the token to be tagged. Rerun the experiment with various alphas and record the results. Which alpha gives the best accuracy and what is the accuracy?

Did the extended feature selector beat the baseline? Intuitively, it should get as least as good accuracy as the baseline. Explain why!

```
In [13]: def better_pos_features(sentence, i, history):
             features = {
                         "suffix(0)": sentence[i],
                         "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

        alphas = [1, 0.5, 0.1, 0.01, 0.001, 0.0001]

        for a in alphas:
            tagger = ScikitConsecutivePosTagger(news_train, features=better_pos_features, clf=
            print(round(tagger.evaluate(news_dev), 4))
```

```
0.8802
0.917
0.9272
0.937
0.9397
0.9416
```

The performance is much better

1.3 Ex 3: Logistic regression (10 points)

1.3.1 Part a.

We proceed with the best feature selector from the last exercise. We will study the effect of the learner. Import *LogisticRegression* and use it with standard settings instead of *BernoulliNB*. Train on *news_train* and test on *news_dev_test* and record the result. Is it better than the best result with Naive Bayes?

```
In [ ]: tagger = ScikitConsecutivePosTagger(news_train, clf=LogisticRegression(solver="lbfgs",
        print(round(tagger.evaluate(news_dev), 4))
```

```
/home/markus/.local/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:460: FutureWa
  "this warning.", FutureWarning)


0.8941
```

1.3.2 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.) Try with C in [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0] and see which value which yields the best result.

Which C gives the best result?

```
In [14]: cs = [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]

         for c in cs:
             tagger = ScikitConsecutivePosTagger(news_train, features=better_pos_features, clf=
             print(round(tagger.evaluate(news_dev), 4))
```

```
0.8331


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
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-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


0.9266


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
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-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


0.9549


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
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-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)


0.9585


/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
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-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
0.9554
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarnin
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-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

```
0.954
```

1.4 Ex 4: Features (10 points)

1.4.1 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 [ ]: best_param = 10
        tagger = ScikitConsecutivePosTagger(
            news_train,
            features=lambda sentence, i, history: {
                "word": sentence[i],
                "prev-word": "<START>" if not i else sentence[i-1],
                "next-word": "<END>" if i == len(sentence)-1 else sentence[i+1],
                "suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]
            },
        clf=LogisticRegression(C=best_param))
        print(round(tagger.evaluate(news_dev), 4))
```

```
0.9668
```

1.4.2 Part b.

Try to add more features to get an even better tagger. Only the fantasy sets limits to what you may consider. Some candidates: is the word a number? Is it capitalized? Does it contain capitals? Does it contain a hyphen? Consider larger contexts? etc. What is the best feature set you can come up with? Train and test various feature sets and select the best one.

If you use sources for finding tips about good features (like articles, web pages, NLTK code, etc.) make references to the sources and explain what you got from them.

Observe that the way *ScikitConsecutivePosTagger.tag()* is written, it extracts the features from a whole sentence before it tags it. Hence it does not support preceding tags as features. It is possible to rewrite *ScikitConsecutivePosTagger.tag()* to extract features after reading each word, and to use the *history* which keeps the preceding tags in the sentence. If you like, you may try it. However, we got surprisingly little gain from including preceding tags as features, and you are not requested to trying it.

```
In [ ]: features = lambda sentence, i, history: {
            "word": sentence[i],
            "prefix(1)": sentence[i][:1],
            "prefix(2)": sentence[i][:2],
            "prefix(3)": sentence[i][:3],
            "prev-prev-word": "<START-START>" if i<1 else sentence[i-2],
            "prev-word": "<START>" if not i else sentence[i-1],
            "next-word": "<END>" if i == len(sentence)-1 else sentence[i+1],
            "next-next-word": "<END-END>" if i>=len(sentence)-2 else sentence[i+2],
            "suffix(1)": sentence[i][-1:],
            "suffix(2)": sentence[i][-2:],
            "suffix(3)": sentence[i][-3:],
        }

        tagger = ScikitConsecutivePosTagger(
            news_train,
            features=features,
            clf=LogisticRegression(C=best_param)
        )

        print(round(tagger.evaluate(news_dev), 4))
```

```
0.9693
```

1.5 Ex5: Larger corpus and evaluation (15 points)

1.5.1 Part a.

We can now test our best tagger so far on the *news_test* set. Do that. How is the result compared to testing on *news_dev_test*?

```
In [ ]: round(tagger.evaluate(news_test), 4)
```

```
Out[ ]: 0.9644
```

1.5.2 Part b.

But we are looking for bigger fish. How good is our settings when trained on a bigger corpus?

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.

Call the Brown corpus with all categories except these three for *rest*. Shuffle the tagged sentences from *rest* and remeber to use the universal pos tagset. Then split the set into 80%-10%-10%: *rest_train*, *rest_dev_test*, *rest_test*.

We can then merge these three sets with the corresponding sets from *news* to get final training and test sets:

- `train = rest_train + news_train`
- `test = rest_test + news_test`

The first we should do is to establish a new baseline. Do this similarly to the way you did for the news corpus above.

```
In [17]: categories = set(brown.categories()) - {'adventure', 'hobbies'}
         brown_sents = brown.tagged_sents(categories=categories, tagset='universal')
         brown_train, brown_dev, brown_test = train_dev_test_split(brown_sents)
```

```
In [ ]: len(brown_train), len(news_train)
```

```
Out[ ]: (38808, 3698)
```

```
In [ ]: baseline = BaselineTagger(brown_train)
        round(baseline.evaluate(brown_dev), 4)
```

```
Out[ ]: 0.8407
```

1.5.3 Part c.

We can then build our tagger for this larger domain. Use the best settings from the earlier exercises, train on *train* and test on *test*. What is the accuracy of your tagger?

Warning: Running this experiment may take 15-30 min.

```
In [ ]: def benchmark_tagger(train_func, train_set, test_set):
            "Trains and tests a tagger"
            train_dur = time.time()
            tagger = train_func(train_set)
            train_dur = time.time() - train_dur

            test_dur = time.time()
            accuracy = tagger.evaluate(test_set)
            test_dur = time.time() - test_dur

            return {
                "tagger": tagger,
```

```
            "accuracy": accuracy,
            "train_dur": train_dur,
            "test_dur": test_dur
        }
```

```
In [ ]: logreg_tagger = lambda train: ScikitConsecutivePosTagger(
            train,
            features=features,
            clf=LogisticRegression(C=best_param)
        )

        results = benchmark_tagger(logreg_tagger, brown_train, brown_test)
        results
```

```
Out[ ]: {'tagger': <__main__.ScikitConsecutivePosTagger at 0x7f558836ef50>,
         'accuracy': 0.9970839235653647,
         'train_dur': 754.7612001895905,
         'test_dur': 22.244492530822754}
```

1.5.4 Part d.

Test the big tagger first on *adventures* then on *hobbies*. Discuss in a few sentences why you see different results from when testing on *test*. Why do you think you got different results on *adventures* from *hobbies*?

```
In [ ]: big_tagger = results["tagger"]

        for c in "hobbies", "adventure":
            print(f"testing on {c}")
            sents = brown.tagged_sents(categories=c, tagset="universal")
            print("Accuracy: ", big_tagger.evaluate(sents))
```

```
testing on hobbies
Accuracy:  0.9659724330560447
testing on adventure
Accuracy:  0.9724409448818898
```

Hobbies and adventures are similar categories, but fairly different from the other ones. They will contain some different terms from the rest of the corpus, and the test set. Categories will trivially determine the difference of the data to some extent, and the test data was part of the same categories as the train data

1.6 Ex6: Comparing to other taggers (10 points)

1.6.1 Part a.

In the lectures, we spent quite some time on the HMM-tagger. NLTK comes with an HMM-tagger which we may train and test on our own corpus. It can be trained by

```
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(news_train)
```

and tested similarly as we have tested our other taggers. Train and test it, first on the *news* set then on the big *train/test* set. How does it perform compared to your best tagger? What about speed?

```
In [ ]: hmm_news = benchmark_tagger(nltk.HiddenMarkovModelTagger.train, news_train, news_test)
        hmm_news

Out[ ]: {'tagger': <HiddenMarkovModelTagger 12 states and 12785 output symbols>,
         'accuracy': 0.9720901436354631,
         'train_dur': 0.22515130043029785,
         'test_dur': 8.102807760238647}

In [ ]: hmm_brown = benchmark_tagger(nltk.HiddenMarkovModelTagger.train, brown_train, brown_tes
        hmm_brown

Out[ ]: {'tagger': <HiddenMarkovModelTagger 12 states and 46264 output symbols>,
         'accuracy': 0.9692914243826091,
         'train_dur': 2.258535385131836,
         'test_dur': 173.38753366470337}
```

The hmm model was much faster on training, but slower when predicting. All in all, it was much faster, but the logistic regression model was more accurate

1.6.2 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:

- `per_tagger = nltk.PerceptronTagger(load=False)`
- `per_tagger.train(train)`

It is tested similarly to our other taggers.

Train and test it, first on the news set and then on the big train/test set. How does it perform compared to your best tagger? Did you beat it? What about speed?

```
In [21]: # I did not have time to run this
         big_per_tagger = nltk.PerceptronTagger(load=False)
         big_per_tagger.train(brown_train)
         big_per_tagger.evaluate(brown_test)


         ---------------------------------------------------------------------------

         KeyboardInterrupt                         Traceback (most recent call last)

         <ipython-input-21-7299e484e977> in <module>()
           1 # I did not have time to run this
           2 big_per_tagger = nltk.PerceptronTagger(load=False)
```

```
----> 3 big_per_tagger.train(brown_train)
      4 big_per_tagger.evaluate(brown_test)


/usr/local/lib/python3.6/dist-packages/nltk/tag/perceptron.py in train(self, sentences
194                     if not guess:
195                         feats = self._get_features(i, word, context, prev, prev2)
--> 196                         guess = self.model.predict(feats)
197                         self.model.update(tags[i], guess, feats)
198                     prev2 = prev


/usr/local/lib/python3.6/dist-packages/nltk/tag/perceptron.py in predict(self, features
 50         scores = defaultdict(float)
 51         for feat, value in features.items():
---> 52             if feat not in self.weights or value == 0:
 53                 continue
 54             weights = self.weights[feat]


KeyboardInterrupt:
```

```
In [20]: per_tagger = nltk.PerceptronTagger(load=False)
         per_tagger.train(news_train)
         per_tagger.evaluate(news_test)
```

```
Out[20]: 0.9608059342421812
```

2 Assignment 2 part B

```
In [ ]: wv = api.load('word2vec-google-news-300')
```

```
/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This fun
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
```

```
In [2]: from itertools import islice
        print(*islice(wv.vocab, 10))
```

```
</s> in for that is on ## The with said
```

```
In [6]: vec_king = wv["king"]
        vec_king.shape
```

```
Out[6]: (300,)
```

2.0.1 Exercise 1 Basics (8 points)

a) How many different words are there in the model? With so many words, how come that the 'cameroon' example fails?

Because cameroon does not exist in the corpus

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

```
In [7]: from math import sqrt
        def norm(v):
            return sqrt((v**2).sum())

        def cosine_between(v, u):
            return (v @ u)/(norm(v) * norm(u))
```

c) Calculate the cosine between the vectors for 'king' and 'queen' and check you get the same as by .similarity('king', 'queen') 2

```
In [9]: vec_queen = wv["queen"]
        cosine_between(vec_king, vec_queen)

Out[9]: 0.651095593291559

In [10]: wv.similarity("king", "queen")

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[10]: 0.6510957
```

yes

2.0.2 Exercise 2 Built in functions (5 points)

There are several built-in functions that let you inspect semantic properties of the embeddings. The most_similar lets you find the nearest neighbor to one or more words. print(wv.most_similar('car', topn=5)) print(wv.most_similar(positive=['car', 'minivan'], topn=5))

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

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

Try a few analogy tests like " king is to man as queen is to ..." " king is to queen as man is to ..." "cat is to kitten as dog is to ..." Add four more examples of your choice. Report the results of the tests. Are the results as expected?

```
In [149]: print(wv.most_similar(positive=['Oslo', 'Sweden'], negative=['Norway'], topn=5))
          print(wv.most_similar(positive=['Puppy', 'Cat'], negative=['Dog'], topn=5))
          print(wv.most_similar(positive=['red', 'red'], negative=['blue'], topn=5))
          print(wv.most_similar(positive=['ecstatic', 'sad'], negative=['happy'], topn=5))
          print(wv.most_similar(positive=['furious', 'scared'], negative=['angry'], topn=5))
          print(wv.most_similar(positive=['psychology', 'society'], negative=['brain'], topn=5
          print(wv.most_similar(positive=['Satan', 'Luke_Skywalker'], negative=['Jesus'], topn=

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


[('Stockholm', 0.7886310815811157), ('Helsinki', 0.648445725440979), ('Stockholm_Sweden', 0.630
[('Kitten', 0.544740617275238), ('Purrfect', 0.4871232211589813), ('Kittens', 0.48437270522117
[('yellow', 0.5963444113731384), ('www.stec_inc.com', 0.4381125867366791), ('white_balls_##-##-
[('heartbreaking', 0.6013249158859253), ('saddening', 0.5985360145568848), ('heartbroken', 0.57
[('terrified', 0.5953051447868347), ('afraid', 0.5589016079902649), ('petrified', 0.5514507293
[('sociology', 0.5582809448242188), ('Sociology', 0.5005502700805664), ('societal', 0.47440478
[('Han_Solo_Chewbacca', 0.531974196434021), ('Darth_Vader', 0.5300464034080505), ('Sith_lords'
```

b) To understand the method a little better, we can try to follow the recipe more directly. Try a = wv['king'] + wv['woman'] - wv['man'] and calculate the cosine between a and the vectors for queen, woman, man, king. You may also calculate the wv.similar_by_vector(a) What does this show regarding how the most_similar works?

```
In [74]: a = wv['king'] + wv['woman'] - wv['man']
         for i in "queen", "woman", "man", "king":
             print(cosine_between(a, wv[i]))

0.7300517397976956
0.3916338704674177
0.12160637424135927
0.8449392050501695


In [67]: wv.similar_by_vector(a)

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[67]: [('king', 0.8449392318725586),
          ('queen', 0.7300517559051514),
          ('monarch', 0.6454660892486572),
          ('princess', 0.6156251430511475),
          ('crown_prince', 0.5818676948547363),
          ('prince', 0.5777117609977722),
          ('kings', 0.5613663792610168),
```

```
('sultan', 0.5376776456832886),
('Queen_Consort', 0.5344247817993164),
('queens', 0.5289887189865112)]
```

The most similar vector is still king and not queen, which counter-argues the analogy hypothesis. man is to king what woman is to king doesnt sound like a good analogy

c) Play around with wv.doesnt_match, e.g. print(wv.doesnt_match(['Norway', 'Denmark', 'Finland', 'Sweden', 'Spain', 'Stockholm'])) Make at least two more examples where the result match human evaluation and two examples where they do not match. Explain!

```
In [154]: groups = [["Trumpet", "Guitar", "Piano", "Mayonnaise", "Drum", "Tuba"],
          ["Captain_America", "Spider-Man", "Batman", "Iron_Man", "Superman", "Jesus"],
          ["Hilter", "Mao", "Stalin", "Kim_Jon_un", "Lenin", "The_Joker"],
          ["square", "triangle", "pentagon", "Michelle_Obama", "circle", "hexagon"]]

          for group in groups:
            print("Which word does not belong of these:")
            print(group)
            print("Model answered", wv.doesnt_match(group), "\n")
```

```
Which word does not belong of these:
['Trumpet', 'Guitar', 'Piano', 'Mayonnaise', 'Drum', 'Tuba']
Model answered Mayonnaise

Which word does not belong of these:
['Captain_America', 'Spider-Man', 'Batman', 'Iron_Man', 'Superman', 'Jesus']
Model answered Jesus

Which word does not belong of these:
['Hilter', 'Mao', 'Stalin', 'Kim_Jon_un', 'Lenin', 'The_Joker']
Model answered Hilter

Which word does not belong of these:
['square', 'triangle', 'pentagon', 'Michelle_Obama', 'circle', 'hexagon']
Model answered square
```

```
/usr/local/lib/python3.6/dist-packages/gensim/models/keyedvectors.py:895: FutureWarning: arrays
  vectors = vstack(self.word_vec(word, use_norm=True) for word in used_words).astype(REAL)
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):
```

2.0.3 Exercise 3 Training a toy model (5 points)

a) Train a word2vec model on the Brown corpus. Follow the recipe from the tutorial, the section Training Your Own Model. You may import the corpus from NLTK by brown.sents().

Beware that this is a toy example. The Brown corpus is too small for training good models. How many times larger is the Google news corpus compared to the Brown corpus?

```
In [139]: wv_brown = gensim.models.Word2Vec(sentences=brown.sents()).wv

In [144]: len(wv.vocab)/len(wv_brown.vocab)

Out[144]: 197.71963355961248
```

The vocabulary of the google corpus is almost 200 times larger than the brown corpus

b) We will compare the Brown model to the 'word2vec-google-news-300'. Try to find the 10 nearest words first to car and then to queen in the two models. What do the examples reveal about the two training corpora?

```
In [145]: wv.similar_by_word('car')

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[145]: [('vehicle', 0.7821096181869507),
           ('cars', 0.7423830032348633),
           ('SUV', 0.7160962820053101),
           ('minivan', 0.6907036304473877),
           ('truck', 0.6735789775848389),
           ('Car', 0.6677608489990234),
           ('Ford_Focus', 0.667320191860199),
           ('Honda_Civic', 0.662684977054596),
           ('Jeep', 0.6511331796646118),
           ('pickup_truck', 0.64414381980896)]

In [147]: wv_brown.similar_by_word("car")

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[147]: [('house', 0.9579110145568848),
           ('room', 0.9165933728218079),
           ('hand', 0.9050480127334595),
           ('hall', 0.9038076996803284),
           ('bed', 0.9008783102035522),
           ('desk', 0.900174081325531),
           ('town', 0.9000574350357056),
           ('road', 0.894163966178894),
           ('corner', 0.8909157514572144),
           ('step', 0.8870737552642822)]

In [146]: wv.similar_by_word("queen")
```

```
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[146]: [('queens', 0.739944338798523),
           ('princess', 0.7070531845092773),
           ('king', 0.6510956883430481),
           ('monarch', 0.6383601427078247),
           ('very_pampered_McElhatton', 0.6357026100158691),
           ('Queen', 0.6163407564163208),
           ('NYC_anglophiles_aflutter', 0.6060680150985718),
           ('Queen_Consort', 0.592379629611969),
           ('princesses', 0.5908075571060181),
           ('royal', 0.5637185573577881)]

In [148]: wv_brown.similar_by_word("queen")

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):


Out[148]: [('veteran', 0.9584357142448425),
           ('surgeon', 0.9573053121566772),
           ('band', 0.9556788802146912),
           ('Sloan', 0.9553066492080688),
           ('minister', 0.9547104835510254),
           ('gown', 0.9546879529953003),
           ('shock', 0.9534328579902649),
           ('gentle', 0.9534325003623962),
           ('calf', 0.9530235528945923),
           ('interview', 0.952724277973175)]
```

The similarities from the google corpus seem more accurate than those of the brown corpus, which supports the theory that more data usually yields better results

c) Inspect the trained Brown model on some of the examples from exercise 2. Does it yield the same results on the analogy tests as the model in exercise 2? 3

```
In [152]: positives = [['Oslo', 'Sweden'],
                       ['Puppy', 'Cat'],
                       ['red', 'red'],
                       ['ecstatic', 'sad'],
                       ['furious', 'scared'],
                       ['psychology', 'society'],
                       ['Satan', 'Luke_Skywalker']]

          negatives = ["Norway", "Dog", "blue", "happy", "angry", "brain", "Jesus"]

          for positive, negative in zip(positives, negatives):
```

```
            try:
              print(wv_brown.most_similar(positive=positive, negative=[negative], topn=5))
            except KeyError as e:
              print(e)
```

```
"word 'Norway' not in vocabulary"
"word 'Puppy' not in vocabulary"
[('green', 0.9342679977416992), ('thin', 0.9279745817184448), ('wide', 0.9259721040725708), ('r
"word 'ecstatic' not in vocabulary"
[('Argiento', 0.8813881874084473), ('unto', 0.8752793073654175), ('hid', 0.8728108406066895),
[('source', 0.9058712124824524), ('recognition', 0.9023469686508179), ('humor', 0.893501400947
"word 'Satan' not in vocabulary"


/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):
```

```
In [155]: groups = [["Trumpet", "Guitar", "Piano", "Mayonnaise", "Drum", "Tuba"],
          ["Captain_America", "Spider-Man", "Batman", "Iron_Man", "Superman", "Jesus"],
          ["Hilter", "Mao", "Stalin", "Kim_Jon_un", "Lenin", "The_Joker"],
          ["square", "triangle", "pentagon", "Michelle_Obama", "circle", "hexagon"]]

          for group in groups:
              print("Which word does not belong of these:")
              print(group)
              try:
                  print("Model answered", wv_brown.doesnt_match(group), "\n")
              except KeyError as e:
                  print(e)
```

```
Which word does not belong of these:
['Trumpet', 'Guitar', 'Piano', 'Mayonnaise', 'Drum', 'Tuba']
Model answered Piano

Which word does not belong of these:
['Captain_America', 'Spider-Man', 'Batman', 'Iron_Man', 'Superman', 'Jesus']
Model answered Jesus

Which word does not belong of these:
['Hilter', 'Mao', 'Stalin', 'Kim_Jon_un', 'Lenin', 'The_Joker']
Model answered Lenin

Which word does not belong of these:
['square', 'triangle', 'pentagon', 'Michelle_Obama', 'circle', 'hexagon']
Model answered square
```

```
/usr/local/lib/python3.6/dist-packages/gensim/models/keyedvectors.py:895: FutureWarning: arrays
  vectors = vstack(self.word_vec(word, use_norm=True) for word in used_words).astype(REAL)
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):
```

In the first example, the results were disasterous, and none of them were the same. In the second one they were ok, as it gave half of the same results.

2.0.4 Exercise 4 Evaluation (5 points)

Gensim comes with several methods for evaluation, and also standard datasets for the tests. Testsets could be found by the tha datapath command, e.g. path=datapath('questions-words.txt') One test you may use is to see how well the model perform on the Google analogy test datset. This can be run by .evaluate_word_analogies(path) Report the key numbers, and try to understand what they mean. To compare 'word2vec-google-news-300' to the Brown embeddings is not too interesting. The difference between them is too large. A test like this becomes more interesting if you try to compare 'word2vec-google-news-300' to e.g. 'glove-wiki-gigaword-300' or you want to inspect the effect of the length of the embeddings by comparing 'glove-wiki-gigaword-300' and 'glove-wiki-gigaword100'.

```
In [120]: gwg3 = api.load("glove-wiki-gigaword-300")

[==================================================-------] 86.4% 324.8/376.1MB downloaded


/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This func
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL


In [119]: gwg1 = api.load("glove-wiki-gigaword-100")

[==================================================] 100.0% 128.1/128.1MB downloaded


/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This func
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL


In [156]: from gensim.test.utils import datapath

          path = datapath("questions-words.txt")

In [162]: evaluation = wv.evaluate_word_analogies(path)

/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This func
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the
  if np.issubdtype(vec.dtype, np.int):
```

```
In [223]: evaluation[0]

Out[223]: 0.7401448525607863

In [224]: evaluation2 = gwg3.evaluate_word_analogies(path)

/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This fun
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of th
  if np.issubdtype(vec.dtype, np.int):


In [228]: evaluation2[0]

Out[228]: 0.7195422354510931

In [225]: evaluation3 = gwg1.evaluate_word_analogies(path)

/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:252: UserWarning: This fun
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of th
  if np.issubdtype(vec.dtype, np.int):


In [229]: evaluation3[0]

Out[229]: 0.6329672585445961
```

The google news corpus scores higher, while the smaller corpora score lower. The score seems to be proportional with the data size. The score evaluate the trueness of the analogy hypothesis in word-embedding vector spaces

2.0.5 Exercise 5 Application (12 points)

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.

a) Train and test a logistic regression classifier as described. Tune the C parameter (regularization, cf. Mandatory 2.A). Report the results from the tuning in a table. How does this classifier perform compared to your results from Mandatory assignment 1?

```
In [168]: from nltk.corpus import movie_reviews
          from gensim.models.doc2vec import Doc2Vec, TaggedDocument

In [205]: from gensim.test.utils import common_texts
          common_texts
```

```
Out[205]: [['human', 'interface', 'computer'],
           ['survey', 'user', 'computer', 'system', 'response', 'time'],
           ['eps', 'user', 'interface', 'system'],
           ['system', 'human', 'system', 'eps'],
           ['user', 'response', 'time'],
           ['trees'],
           ['graph', 'trees'],
           ['graph', 'minors', 'trees'],
           ['graph', 'minors', 'survey']]

In [207]: try:
              raw_movie_docs = [(movie_reviews.words(fileid), category) for
                                category in movie_reviews.categories() for fileid in
                                movie_reviews.fileids(category)]
          except LookupError:
              nltk.download('movie_reviews')
              raw_movie_docs = [(movie_reviews.words(fileid), category) for
                                category in movie_reviews.categories() for fileid in
                                movie_reviews.fileids(category)]

In [208]: random.seed(2405)
          random.shuffle(raw_movie_docs)
          movie_test = raw_movie_docs[:200]
          movie_dev  = raw_movie_docs[200:]
          train_data, dev_test_data = movie_dev[200:], movie_dev[:200]

In [209]: print(train_data[0])

(['ugh', '.', 'that', 'about', 'sums', 'this', 'movie', ...], 'neg')


In [210]: train_texts, train_target = map(list, zip(*train_data))
          dev_test_texts, dev_test_target = map(list, zip(*dev_test_data))

In [211]: documents = [TaggedDocument(doc, [i]) for i, doc in enumerate(train_texts)]
          model = Doc2Vec(documents, vector_size=5, window=2, min_count=1, workers=4)

In [212]: train_vectors = [model.infer_vector(doc) for doc in train_texts]
          dev_test_vectors = [model.infer_vector(doc) for doc in dev_test_texts]

In [213]: cs = [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
          fitting_results = np.zeros((6, 2))
          for i, c in enumerate(cs):
              clf = LogisticRegression(C=c)
              clf.fit(train_vectors, train_target)
              fitting_results[i, 0] = c
              fitting_results[i, 1] = clf.score(dev_test_vectors, dev_test_target)

In [214]: import pandas as pd
          result1 = pd.DataFrame(fitting_results, columns=["C", "accuracy"])
```

```
Out[214]:          C  accuracy
          0     0.01     0.635
          1     0.10     0.640
          2     1.00     0.630
          3    10.00     0.630
          4   100.00     0.630
          5  1000.00     0.630
```

The results are pretty bad. Count vectorization still seems way better

b) In this task, one could either use the document vectors directly, or normalize the length of each document vector to unit length before classifying. Try both options and compare the results.

```
In [215]: def normalize(x):
              norm = np.linalg.norm(x)
              return x if norm==0 else x/norm

          normalized_train = list(map(normalize, train_vectors))
          normalized_dev_test = list(map(normalize, dev_test_vectors))

In [216]: fitting_results2 = np.zeros((6, 2))
          for i, c in enumerate(cs):
              clf = LogisticRegression(C=c)
              clf.fit(normalized_train, train_target)
              fitting_results2[i, 0] = c
              fitting_results2[i, 1] = clf.score(normalized_dev_test, dev_test_target)

In [217]: result2 = pd.DataFrame(fitting_results2, columns=["C", "accuracy"])

Out[217]:          C  accuracy
          0     0.01     0.605
          1     0.10     0.630
          2     1.00     0.615
          3    10.00     0.615
          4   100.00     0.605
          5  1000.00     0.600
```

Slightly worse than without normalization

c) In general, embeddings are used together with neural networks. Scikit-learn provides a simple multi-layered neural network classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html Rerun the classification experiment with this classifier. Try various activation functions and report the results for the various activation functions in a table. Warning! When using neural networks and embeddings for word classification, you would use more elaborate models than the simple bag-of-words model, e.g. convolutional networks or recurrent networks which take word order into consideration. END of Mandatory 2

In [218]:
```
from sklearn.neural_network import MLPClassifier
fitting_results3 = np.zeros((3, 1))
activations = ("relu", "logistic", "tanh")

for i, activation in enumerate(activations):
    nn = MLPClassifier(activation=activation, max_iter=1000)
    nn.fit(train_vectors, train_target)
    fitting_results3[i, 0] = nn.score(dev_test_vectors, dev_test_target)

pd.DataFrame(fitting_results3.T, columns=activations)
```

Out[218]:

	relu	logistic	tanh
0	0.615	0.63	0.63