IN4080_2020_Mandatory_2_A

September 25, 2020

1 IN4080, 2020, Mandatory assignment 2, part A

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.

[44]:
```
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]
```

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*.

[2]:
```
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
```

[3]:
```
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.

[4]:
```
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?

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?

Deliveries: Code and results of runs for both parts. For both parts, also answers to the questions.

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.

[48]:
```
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?

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?

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!

Deliveries: Code, results of runs, answers to questions.

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?

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?

Deliveries: Code. Results of the runs. Answers to the questions.

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.

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.

Deliveries: Code. Results of the runs. Answers to the questions.

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*?

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.

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.

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*?

Deliveries: Code. Results of the runs. Answers to the questions.

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?

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?

Deliveries: Code. Results of the runs. Answers to the questions.

2 End of part A. Continue to part B