in4080_2019_mandatory_2_b

October 13, 2019

1 IN4080, 2019, Mandatory assignment B

This is part B. Do part A first.

See part A for delivery date and general requirements

2 Part B

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
- 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 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 setting 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 more than an hour. So make sure you start the work early enough. (You might do other things while the experiments run.)

2.0.1 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.

[29]:
```
import re
import pprint
import nltk
from nltk.corpus import brown
import pandas as pd
import numpy as np
import random

import sklearn
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*.

[30]:
```
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
```

[31]:
```
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.

[32]:
```
tagger = ConsecutivePosTagger(train_sents)
print(round(tagger.evaluate(test_sents), 4))
```

```
0.7915
```

This should give results comparable to the NLTK book.

2.1 Ex 1: initial experiment and baseline (10 points)

2.1.1 Part a. Some simple refinements

The Brown tags that come with NLTK are extended compared to the original tag set. To get better results from our tagger, we strip off the part after the hyphen and stick to the original Brown tags, a tagset of 87 tags. (This is somewhat simplified, cf., weekly exercise set 2.) We can repeat the training and testing, and we see a slightly improved result.

[33]:
```
def originize(tagged_sents):
    """Change tags to original Brown tags in tagged_sents"""
    return [ [(word, tag.split('-')[0]) for (word,tag) in sent]
             for sent in tagged_sents]

orig_train_sents = originize(train_sents)
orig_test_sents = originize(test_sents)

orig_tagger_1 = ConsecutivePosTagger(orig_train_sents)
print(round(orig_tagger_1.evaluate(orig_test_sents), 4))
```

```
0.8314
```

We will use these original Brown tags for the rest of this exercise set.

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*

And we will use the original Brown tags as explained.

- 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.

[34]:
```
complete_set = train_sents + test_sents
news_train, news_dev_test, news_test = np.split(complete_set, [int(.
↪8*len(complete_set)), int(.9*len(complete_set))])

news_train = originize(news_train)
news_dev_test = originize(news_dev_test)
news_test = originize(news_dev_test)

conpos_tagger = ConsecutivePosTagger(news_train)
print(round(conpos_tagger.evaluate(news_dev_test), 4))
```

```
0.8007
```

2.1.2 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 he NLTK-tagger beat the baseline?

[35]:
```
from nltk import ConditionalFreqDist
class BaselinePosTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        self.train_sents = train_sents
        self.cfd = ConditionalFreqDist([(word.lower(),tag) for sentence in␣
↪train_sents for (word,tag) in sentence])
        self.max_ = 0
        self.most_common_tag = ''
        self.most_common_pos()

    def most_common_pos(self):
        tags = {}
        max_  = 0
        max_word = {}
        for sentence in self.train_sents:
            for word, tag in sentence:
                tags[word] = tag
        for key in self.cfd:
            if self.cfd[key].N() > max_:
```

```
                max_ = self.cfd[key].N()
                max_word[max_] = key
        self.max = max_
        self.most_common_tag = tags[max_word[max_]]


    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            if self.cfd[word].N() > 0:
                history.append(self.cfd[word].max())
            else:
                history.append(self.most_common_tag)
        return zip(sentence, history)
```

[36]:
```
tagger = BaselinePosTagger(news_train)
print(round(tagger.evaluate(news_dev_test), 4))
```

```
0.7434
```

NLTK-tagger beats my baseline

2.2 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.

[37]:
```
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(alpha=0.5)):
        # Using pos_features as default.
        # Using BernoulliNB() (with alpha/lidstone 0.5)
        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 get_features(self):
        return self.dict.vec.get_feature_names()

    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)
```

2.2.1 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 original NLTK?

[40]:
```
SciKit_tagger = ScikitConsecutivePosTagger(news_train)
print(round(SciKit_tagger.evaluate(news_dev_test), 4))
```

```
0.7258
```

I do not get the same results as ConsecutivePosTagger maybe it is because that we use BernoulliNB

2.2.2 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. 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 NLTK, worse results or better results?

[11]:
```
alphas = [1, 0.5, 0.1, 0.01, 0.001, 0.0001]
lst = []
for alpha in alphas:
```

```
    SciKit_tagger =␣
 ↪ScikitConsecutivePosTagger(news_train,features=pos_features, clf =␣
 ↪BernoulliNB(alpha=alpha))
    lst.append(round(SciKit_tagger.evaluate(news_dev_test), 4))
df = pd.DataFrame(lst, index=alphas,columns = ['alpha'])
df
```

[11]:

	alpha
1.0000	0.6476
0.5000	0.7258
0.1000	0.8039
0.0100	0.8102
0.0010	0.8095
0.0001	0.8116

2.2.3 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!

[12]:
```
lst = []
def extended_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]
         features["word"] = sentence[i]
     return features

for alpha in alphas:
    SciKit_tagger =␣
 ↪ScikitConsecutivePosTagger(news_train,features=extended_features, clf =␣
 ↪BernoulliNB(alpha=alpha))
    lst.append(round(SciKit_tagger.evaluate(news_dev_test), 4))
df = pd.DataFrame(lst, index=alphas,columns = ['alpha'])
df
```

[12]:
	alpha
1.0000	0.6353
0.5000	0.7524
0.1000	0.8631
0.0100	0.8765
0.0010	0.8822
0.0001	0.8882

a way to calculate a features contribution of a feature f toward the label likelihood for a label is P(f | label) = count(f, label) / count(label)

However, this simple approach can become problematic when a feature never occurs with a given label in the training set. In this case, our calculated value for P(f | label) will be zero, which will cause the label likelihood for the given label to be zero. Thus, the input will never be assigned this label, regardless of how well the other features fit the label.

applying an aplha value will a to the probabillity and preventing this. This is know as smoothing.

2.3 Ex 3: Logistic regression (5 points)

2.3.1 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?

[13]:
```
from sklearn.linear_model import LogisticRegression
import warnings
from sklearn.exceptions import ConvergenceWarning
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=ConvergenceWarning)

def log_scikitConsecutivePosTagger(C=1, features=extended_features):
    SciKit_tagger = ScikitConsecutivePosTagger(news_train,
                                               features=features,
                                               clf = LogisticRegression(C=C,␣
 ↪n_jobs=-1, solver='lbfgs'))

    return round(SciKit_tagger.evaluate(news_dev_test), 4)

print(log_scikitConsecutivePosTagger())
```

```
/home/peder/anaconda3/envs/IN4080/lib/python3.7/site-
packages/sklearn/linear_model/logistic.py:469: FutureWarning: Default
multi_class will be changed to 'auto' in 0.22. Specify the multi_class option to
silence this warning.
  "this warning.", FutureWarning)

0.9032
```

2.3.2 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] and see which value which yields the best result.

Which C gives the best result?

[14]:
```
C_ = [0.01, 0.1, 1.0, 10.0, 100.0]
lst = []
for C in C_:
    lst.append(log_scikitConsecutivePosTagger(C))
df = pd.DataFrame(lst, index=C_,columns = ['C'])
df
```

[14]:

	C
0.01	0.7246
0.10	0.8516
1.00	0.9032
10.00	0.9136
100.00	0.9123

The problem comes when you have a lot of parameters (a lot of independent variables) but not too much data. In this case, the model will often fit the data almost perfectly. However because these characteristic don't appear in future data you see, the model predicts poorly.

to solve this, you add penalizes large values of the parameters by minimizing. This is also know as regularization

smaller values of c specify stronger regularization

2.4 Ex 4: Features (10 points)

2.4.1 Part a.

We will now stick to the LogistiRegression() with the optimal C from the last point and see whether we may 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

[15]:
```
def extended_features_2(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]
        features["word"] = sentence[i]
        if i < len(sentence) - 1:
            features["next-word"] = sentence[i+1]
    return features
```

```
print(log_scikitConsecutivePosTagger(10,extended_features_2))
```

```
0.9252
```

2.4.2 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.

[16]:
```
suffix_fdist = nltk.FreqDist()
common_suffixes = [suffix for (suffix, count) in suffix_fdist.most_common(100)]
for word in brown.words():
    word = word.lower()
    suffix_fdist[word[-1:]] += 1
    suffix_fdist[word[-2:]] += 1
    suffix_fdist[word[-3:]] += 1

def extended_features_fantasy(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:],
                "prefix(1)": sentence[i][0],
                "prefix(2)": sentence[i][:2],
                "prefix(3)": sentence[i][:3]}
    for word in sentence:
        for suffix in common_suffixes:
            features['endswith({})'.format(suffix)] = word.lower().
 ↪endswith(suffix)
    if i == 0:
        features["prev-word"] = "<START>"
        features["prev-tag"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
        features["word"] = sentence[i]
        features['prev-word-is-one-char']: len(sentence[i-1]) == 1
        features['prev-word-capitalized'] = sentence[i-1].isupper(),
        features['prev-word-is-lower'] = sentence[i-1].islower(),
```

```
        features['prev-word-is-hyphen'] = sentence[i-1] == '-'
        features['prev-word-is-isalpha'] = sentence[i-1].isalpha()
        features['prev-word-is-digit'] = sentence[i-1].isdigit()
        if i < len(sentence) - 1:
            features["next-word"] = sentence[i+1]
            features['next-word-capitalized'] = sentence[i+1].isupper(),
            features['next-word-is-one-char'] = len(sentence[i+1]) == 1
            features['next-word-is-lower'] = sentence[i+1].islower(),
            features['next-word-is-end-of-sentence'] = sentence[i+1] == '.'
    return features
```

[17]:
```
scikit_tagger = ScikitConsecutivePosTagger(news_train,
                                           features=extended_features_fantasy,
                                           clf = LogisticRegression(C=10,␣
 ↪n_jobs=-1, solver='lbfgs'))
print(round(scikit_tagger.evaluate(news_dev_test), 4))
```

```
0.9407
```

As we can see the added features help to make a better model. I tried to extract the most important features but I could not get it to work. It would have been interesting to see what feature gave the largest contribution. I add prefix for sentences - that had a positive impact on the models preformance.

2.5 Ex5: Larger corpus and evaluation (10 points)

2.5.1 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*?

[18]:
```
print(round(scikit_tagger.evaluate(news_test), 4))
```

```
0.9407
```

2.5.2 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 strip the tags down to the original Brown tags. 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.

[19]:
```
cat = [cat for cat in brown.categories() if cat != 'adventure' and cat !=␣
 ↪'hobbies' and cat!= 'news']
rest = list(brown.tagged_sents(categories=cat))
random.seed(2920)
random.shuffle(rest)
```

[20]:
```
rest_train, rest_dev_test, rest_test = np.split(rest, [int(.8*len(rest)), int(.
 ↪9*len(rest))])

rest_train = originize(rest_train)
rest_dev_test = originize(rest_dev_test)
rest_test = originize(rest_test)

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

[21]:
```
baseline_tagger = ScikitConsecutivePosTagger(rest_train, clf =␣
 ↪LogisticRegression(n_jobs=-1, solver='lbfgs'))
print(round(baseline_tagger.evaluate(rest_test), 4))
```

```
0.865
```

2.5.3 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 30-60 min.

[22]:
```
scikit_tagger = ScikitConsecutivePosTagger(train,
                                           features=extended_features_fantasy,
                                           clf = LogisticRegression(C=10,␣
 ↪n_jobs=-1, solver='lbfgs'))
print(round(scikit_tagger.evaluate(test), 4))
```

```
0.9677
```

2.5.4 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*?

[23]:
```
hob = brown.tagged_sents(categories='hobbies')
hob = originize(hob)
print(round(scikit_tagger.evaluate(hob), 4))
```

```
0.952
```

[24]:
```
adventures = brown.tagged_sents(categories='adventure')
adventures = originize(adventures)
print(round(scikit_tagger.evaluate(adventures), 4))
```

```
0.9605
```

I think the reason why we get a better result on adventures dataset than hobbies is because the adventure data set is more similar to the other categories than hobbies dataset.

2.6 Ex6: Comparing to other taggers (5 points)

2.6.1 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?

[44]:
```
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(news_train)
print(round(news_hmm_tagger.evaluate(news_test), 4))
```

```
0.8543
```

[45]:
```
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(train)
print(round(news_hmm_tagger.evaluate(test), 4))
```

```
0.9422
```

It preforms good compared to my tagger. The hiddenMarkovModelTagger lack the ability to produce context-dependent content since they cannot take into account the full chain of prior states.

2.6.2 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:

[26]:
```
%%time
per_tagger = nltk.PerceptronTagger(load=False)
per_tagger.train(news_train)
```

```
CPU times: user 18.3 s, sys: 0 ns, total: 18.3 s
Wall time: 18.3 s
```

[27]:
```
print(round(per_tagger.evaluate(news_test), 4))
```

```
0.9319
```

[28]:
```
%%time
per_tagger = nltk.PerceptronTagger(load=False)
per_tagger.train(train)
print(round(per_tagger.evaluate(test), 4))
```

```
0.9712
CPU times: user 3min 33s, sys: 0 ns, total: 3min 33s
Wall time: 3min 33s
```

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?

0.9605 compared to 0.9712 is pretty close but I did not beat it. If I would have added som more feature maybe I would have gotten a better result. The perceptrontagger is faster than my model even if I use all cores on my cpu.

2.7 End of mandatory assignment 2 IN4080 2019