mandatory_2_in4080

October 9, 2022

1 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.

[37]:
```
import re
import pprint
import nltk
nltk.download("brown")
nltk.download('universal_tagset')
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]
```

```
[nltk_data] Downloading package brown to /root/nltk_data…
[nltk_data]   Unzipping corpora/brown.zip.
[nltk_data] Downloading package universal_tagset to /root/nltk_data…
[nltk_data]   Unzipping taggers/universal_tagset.zip.
```

[38]:
```
def pos_features(sentence, i, history):
  features = {
      "suffix(1)": sentence[i][-1:],
      "suffix(2)": sentence[i][-2:],
      "suffix(3)": sentence[i][-3:]
  }
  if i == 0:
    features["prev-word"] = "<START>"
  else:
    features["prev-word"] = sentence[i-1]
  return features
```

[ ]:
```
class ConsecutivePosTagger(nltk.TaggerI):
  def __init__(self, train_sents, features=pos_features):
    self.features = features
    train_set = []
```

```
    for tagged_sent in train_sents:
      untagged_sent = nltk.tag.untag(tagged_sent)
      history = []

      for i, (word, tag) in enumerate(tagged_sent):
        featureset = features(untagged_sent, i, history)
        train_set.append( (featureset, tag) )
        history.append(tag)

    self.classifier = nltk.NaiveBayesClassifier.train(train_set)

  def tag(self, sentence):
    history = []
    for i, word in enumerate(sentence):
      featureset = self.features(sentence, i, history)
      tag = self.classifier.classify(featureset)
      history.append(tag)
    return zip(sentence, history)
```

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

```
0.7915
```

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

1.1.1 1.1.1 Part a. Tag set and experimental set-up

We will simplify and use the **universal pos tagset** in this exercise. The main reason is that it makes the experiments run faster. With more time, it would have been interesting to also run all the experiments with the original set of tags and compare the effect the tag set has on the results.

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 in the introduction? Why do you think one of the tagsets yields higher scores than the other one?

[39]:
```
size = int(len(tagged_sents) * 0.1)
tagged_sents_universial = brown.tagged_sents(categories='news',␣
 ↪tagset="universal")
train_dev, news_test = tagged_sents_universial[size:], tagged_sents_universial[:
 ↪size]
news_train, news_dev_test = tagged_sents_universial[size:],␣
 ↪tagged_sents_universial[:size]
```

[ ]:
```
tagger_uni = ConsecutivePosTagger(news_train)
print(round(tagger_uni.accuracy(news_dev_test), 4))
```

```
0.873
```

The results are stronger when I train and test a model on the smaller tagset. This is because it's easier to distinguish between fewer classes.

1.1.2 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 together with the universal tags beat the baseline?

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

[ ]:
```
class MostFreqClassBaseline(nltk.TaggerI):
  def __init__(self, train_data) -> None:
    self.class_count = {} # {word -> {tag1 : x1, tag2 : x2, ...}}
    self.most_freq_tag_mapper = {} # for unknown words

    for sentence in train_data:
      for word, tag in sentence:
        # add all tags to most_freq_tag
        if tag in self.most_freq_tag_mapper:
          self.most_freq_tag_mapper[tag] += 1
        else:
          self.most_freq_tag_mapper[tag] = 1

        if word in self.class_count:
          if tag in self.class_count[word]:
            # +1 for this tag
            self.class_count[word][tag] += 1
          else:
```

```
            # add new tag to word
            self.class_count[word][tag] = 1
        else:
          # new word and tag
          self.class_count[word] = {tag : 1}

    for word in self.class_count:
      self.class_count[word] = max(
          self.class_count[word], key=self.class_count[word].get
      )

    self.most_freq_tag = max(self.most_freq_tag_mapper,
                             key=self.most_freq_tag_mapper.get)

  def tag(self, sentence):
    history = []
    for word in sentence:
      try:
        tag = self.class_count[word]
      except KeyError:
        tag = self.most_freq_tag
      history.append(tag)
    return zip(sentence, history)
```

[ ]:
```
baseline = MostFreqClassBaseline(news_train)
print(round(baseline.accuracy(news_dev_test), 4))
```

```
0.9312
```

The tagger from part (a) doesn't beat the baseline.

1.1.3 1.2 Ex 2: 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 BernoulliNBclassifier which should correspond to the way it is done in NLTK.

[40]:
```
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?

[ ]:
```
tagger_scikit = ScikitConsecutivePosTagger(news_train)
print(round(tagger_scikit.accuracy(news_dev_test), 4))
```

```
0.8635
```

This model is alot stronger than the model from 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 5 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?

```
[ ]: for alpha in [1, 0.5, 0.1, 0.01, 0.001, 0.0001]:
       tagger_scikit = ScikitConsecutivePosTagger(
           news_train, clf = BernoulliNB(alpha = alpha))
       acc = round(tagger_scikit.accuracy(news_dev_test), 4)
       print(f"alpha = {alpha} | accuracy = {acc}")
```

```
alpha = 1 | accuracy = 0.8635
alpha = 0.5 | accuracy = 0.8796
alpha = 0.1 | accuracy = 0.8754
alpha = 0.01 | accuracy = 0.8721
alpha = 0.001 | accuracy = 0.8703
alpha = 0.0001 | accuracy = 0.8694
```

The alpha that yields the best result on the dev set is alpha = 0.5.

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!

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

```
[ ]: for alpha in [1, 0.5, 0.1, 0.01, 0.001, 0.0001]:
       tagger_scikit = ScikitConsecutivePosTagger(
           news_train, clf = BernoulliNB(alpha = alpha),
           features = pos_features_with_word)
       acc = round(tagger_scikit.accuracy(news_dev_test), 4)
       print(f"alpha = {alpha} | accuracy = {acc}")
```

```
alpha = 1 | accuracy = 0.8957
alpha = 0.5 | accuracy = 0.9214
alpha = 0.1 | accuracy = 0.9288
alpha = 0.01 | accuracy = 0.9346
```

```
alpha = 0.001 | accuracy = 0.9382
alpha = 0.0001 | accuracy = 0.9386
```

The best results are achieved when using an alpha = 0.0001, with an accuracy of 0.94. The model with the extended feature set got a slightly better result compared to the baseline. This is expected because the model should atleast be able to learn what tag is most frequent when the word itself is used as a feature.

1.1.4 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?

[ ]:
```
tagger_scikit = ScikitConsecutivePosTagger(
      news_train, clf = LogisticRegression(),
      features = pos_features_with_word)
acc = round(tagger_scikit.accuracy(news_dev_test), 4)
print(f"LogReg: accuracy = {acc}")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

LogReg: accuracy = 0.956
```

The result is alot better with LogisicRegression compared to NaiveBayes.

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 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.

[ ]:
```
for C in [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]:
  tagger_scikit = ScikitConsecutivePosTagger(
      news_train, clf = LogisticRegression(C = C),
```

```
        features = pos_features_with_word)
    acc = round(tagger_scikit.accuracy(news_dev_test), 4)
    print(f"C = {C} | accuracy = {acc}")
```

```
C = 0.01 | accuracy = 0.8532

/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

C = 0.1 | accuracy = 0.9272

/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

C = 1.0 | accuracy = 0.956

/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

C = 10.0 | accuracy = 0.958

/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
```

```
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

```
C = 100.0 | accuracy = 0.9574
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

```
C = 1000.0 | accuracy = 0.956
```

The best results are achieved when C = 10.0.

1.1.5 1.4 Ex 4: Features (15 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.

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

  if i == len(sentence) - 1:
    features["next_word"] = "<EOS>"
  else:
    features["next_word"] = sentence[i+1]
  return features
```

[ ]:
```python
tagger_scikit = ScikitConsecutivePosTagger(
      news_train, clf = LogisticRegression(C=100),
```

```
      features = feature_two_words)
acc = round(tagger_scikit.accuracy(news_dev_test), 4)
print(f"LogReg: accuracy = {acc}")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

LogReg: accuracy = 0.9672
```

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. Expect, however, that the tagger will become much slower. 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.

[ ]:
```
def feature_set(sentence, i, history, puncts = [ ".", ",", "!", "?", ";", ":",␣
 ↪"(", ")", "``"]):
  features = {
      "suffix(1)": sentence[i][-1:],
      "suffix(2)": sentence[i][-2:],
      "suffix(3)": sentence[i][-3:],
      "word" : sentence[i],
      "word_lower" : sentence[i].lower(),
      "capitalized" : sentence[i][0].isupper(),
      "numeric" : sentence[i].isnumeric(),
      "punct" : sentence[i] in puncts,
      "contains_hyphen" : "-" in sentence[i]
  }
  if i == 0:
    features["prev-word"] = "<START>"
```

```
  else:
    features["prev-word"] = sentence[i-1]
    features["prev_word_lower"] : sentence[i-1].lower()
    features["prev_capitalized"] = sentence[i-1][0].isupper()
    features["prev_numeric"] = sentence[i-1].isnumeric()
    features["prev_punct"] = sentence[i-1] in puncts
    features["prev_contains_hyphen"] = "-" in sentence[i-1]

  if i == len(sentence) - 1:
    features["next_word"] = "EOS"
  else:
    features["next_word"] = sentence[i+1]
    features["next_word_lower"] : sentence[i+1].lower()
    features["next_capitalized"] = sentence[i+1][0].isupper()
    features["next_numeric"] = sentence[i+1].isnumeric()
    features["next_punct"] = sentence[i+1] in puncts
    features["next_contains_hyphen"] = "-" in sentence[i + 1]
  return features

# https://towardsdatascience.com/pos-tagging-using-crfs-ea430c5fb78b
# got: word_lower and punct
```

[ ]:
```
tagger_scikit2 = ScikitConsecutivePosTagger(
      news_train, clf = LogisticRegression(),
      features = feature_set)

acc = round(tagger_scikit2.accuracy(news_dev_test), 4)
print(f"LogReg: accuracy = {acc}")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

LogReg: accuracy = 0.9691
```

1.1.6 1.5 Ex 5: Training on a larger corpus (15 points)

1.5.1 Part a. We have so far used a smaller corpus, the news section, for finding optimal settings for a tagger. We will now try to make a better tagger by training on a larger 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 remember 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**
- **dev_test = rest_dev_test + uni_news_dev_test**
- **test = rest_test + news_test**

Prepare the corpus as described.

[41]:
```
cats4train = [
    # 'adventure',
    'belles_lettres',
    'editorial',
    'fiction',
    'government',
    # 'hobbies',
    'humor',
    'learned',
    'lore',
    'mystery',
    # 'news',
    'religion',
    'reviews',
    'romance',
    'science_fiction'
]

rest = [sent for sent in brown.tagged_sents(categories=cats4train,␣
 ↪tagset="universal")]
```

[42]:
```
# inplace shuffle
np.random.shuffle(rest)
```

[43]:
```
idx = int(len(rest) * 0.1)
rest_train_dev, rest_test = rest[idx:], rest[:idx]
rest_train, rest_dev = rest_train_dev[idx:], rest_train_dev[:idx]
```

[44]:
```
test = rest_test + news_test
train = rest_train + news_train
dev = rest_dev + news_dev_test
```

1.5.2 Part b. The next step is to establish a baseline for a tagger trained on this larger corpus, and evaluate it on dev_test.

[ ]:
```
baseline = MostFreqClassBaseline(train)
print(round(baseline.accuracy(dev), 4))
```

```
0.9453
```

1.5.3 Part c. We can then train our tagger on this larger corpus. Use the best settings from the earlier exercises, train on train and test on dev_test. What is the accuracy of your tagger?

[ ]:
```
tagger_scikit_entire_brown = ScikitConsecutivePosTagger(
      train,
      clf = LogisticRegression(
          #C=100,
          # solver="saga",
          max_iter = 100,
          # penalty = "none"
      ),
      features = feature_set)

acc = round(tagger_scikit_entire_brown.accuracy(dev), 4)
print(f"LogReg: accuracy = {acc}")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:818:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

```
LogReg: accuracy = 0.9764
```

The accuracy is 0.98.

###**1.6 Ex6: Evaluation metrics (10 points)** ####**1.6.1 Part a.** The accuracy should be quite decent now > 0.97. Still, we will like to find out more about where the tagger makes mistakes. With only 12 different tags, we can get all the results into a confusion table. Take a look at https://www.nltk.org/api/nltk.tag.api.html and make a confusion table for the results on dev_test. Make sure you understand what the rows and columns are.

[ ]:
```
conf_mat = tagger_scikit_entire_brown.confusion(dev)
```

[ ]:
```
print(conf_mat)
```

```
     |                           C          N          P          V       |
     |          A     A     A     O     D     O     N     R     P     E       |
     |          D     D     D     N     E     U     U     O     R     R       |
     |     .    J     P     V     J     T     N     M     N     T     B     X |
-----+-------------------------------------------------------------------------+
   . |<12701>    .     .     .     .     .     .     .     .     .     .     . |
 ADJ |     . <6908>    4   170     .     .   256     3     .     .    90     . |
 ADP |     1     3<12535>   45     5    19     5     1     5   168    10     . |
 ADV |     .   156    84 <4457>    7    19    34     .     .    33     6     . |
CONJ |     .     .     4     5 <3143>    1     .     .     .     .     1     . |
 DET |     .     .    23     6     1<11989>    2     .    24     .     .     . |
NOUN |     .   255     .    17     .     2<23837>   39     2     2   223     3 |
 NUM |     .     3     .     .     .     .    19 <1237>     .     .     .     . |
PRON |     .     .    44     1     .    22     5     . <4169>     .     .     . |
 PRT |     .     3    94    23     .     .    24     .     1 <2428>     8     . |
VERB |     .    34    10    14     .     .   272     .     1     .<15562>     . |
   X |     .     6     .     1     .     .    68     .     1     .     3   <19>|
-----+-------------------------------------------------------------------------+
(row = reference; col = test)
```

####**1.6.2 Part b.** Finding hints on the same web page, calculate the precision, recall and f-measure for each tag and display the results in a table.

[36]:
```
def print_table(metric, metric_dict):
  print(metric)
  print("-"*13)
  for k,v in metric_dict.items():
    print(f"{k:<4}", ":", str(round(v,4)))
```

[ ]:
```
prec = tagger_scikit_entire_brown.precision(dev)
```

[31]:
```
print_table("precision", prec)
```

```
precision
-------------
.    : 0.9999
ADJ  : 0.9376
ADP  : 0.9794
ADV  : 0.9405
CONJ : 0.9959
DET  : 0.9948
NOUN : 0.9721
NUM  : 0.9664
PRON : 0.9919
PRT  : 0.9228
VERB : 0.9786
X    : 0.8636
```

[ ]:
```
recall = tagger_scikit_entire_brown.recall(dev)
```

[33]:
```
print_table("recall", recall)
```

```
recall
-------------
.    : 1.0
ADJ  : 0.9296
ADP  : 0.9795
ADV  : 0.9293
CONJ : 0.9965
DET  : 0.9954
NOUN : 0.9777
NUM  : 0.9825
PRON : 0.983
PRT  : 0.9407
VERB : 0.9792
X    : 0.1939
```

[ ]:
```
f_measure = tagger_scikit_entire_brown.f_measure(dev)
```

[35]:
```
print_table("f_measure", f_measure)
```

```
f_measure
-------------
.    : 1.0
ADJ  : 0.9336
ADP  : 0.9795
ADV  : 0.9349
CONJ : 0.9962
DET  : 0.9951
NOUN : 0.9749
NUM  : 0.9744
PRON : 0.9874
PRT  : 0.9317
VERB : 0.9789
X    : 0.3167
```

####**1.6.3 Part c.** Calculate the macro precision, macro recall and macro f-measure across the 12 tags.

[ ]:
```
# macro precision:
macro_prec = sum(prec.values())/12
print(f"Macro precision: {macro_prec:.3f}")
```

```
Macro precision: 0.962
```

[ ]:
```
# macro precision:
macro_recall = sum(recall.values())/12
print(f"Macro recall: {macro_recall:.3f}")
```

```
Macro recall: 0.907
```

[ ]:
```
# macro precision:
macro_f = sum(f_measure.values())/12
print(f"Macro f-measure: {macro_f:.3f}")
```

```
Macro f-measure: 0.917
```

1.1.7 1.7 Ex 7: Error analysis (10 points)

Sometimes when we make classifiers for NLP phenomena, it makes sense to inspect the errors more thoroughly. Where does the classifier make errors? What kind of errors? Find five sentences where at least one token is mis-classified, and display these sentences on the follwing form, with the pred(icted) and gold tags.

Identify the words that are tagged differently. Comment on each of the differences. Would you say that the predicted tag is wrong? Or is there a genuine ambiguity such that both answers are defendable? Or is even the gold tag wrong?

[ ]:
```
errors = []

for sent in dev:
  sent2tag = [i[0] for i in sent]
  tags = [i[1] for i in sent]
  sent_preds = list(tagger_scikit_entire_brown.tag(sent2tag))
  preds = [i[1] for i in sent_preds]
  eq = (preds == tags)
  if not eq:
    info = (sent, sent_preds)
    errors.append(info)

  if len(errors) == 5:
    break
```

[ ]:
```
def error_parser(error_iter):
  print("Token          pred   gold")
  print("==========================")
  for i in range(len(error_iter[0])):
    s = f"{error_iter[0][i][0]}".ljust(15)
    print(s, end="")
    s2 = f"{error_iter[1][i][1]}"
    s3 = f"{error_iter[0][i][1]}"
    print(s2.ljust(5), end=" ")
```

```
    print(s3)
```

[ ]:
```
error_iter = iter(errors)
error_parser(next(error_iter))
```

```
Token          pred   gold
==========================
A              DET    DET
seeping        VERB   VERB
coldness       NOUN   NOUN
entered        VERB   VERB
Holden's       NOUN   NOUN
being          VERB   NOUN
;              .      .
;              .      .
```

The classifier is definetly wrong, but I consider this to be an honest mistake. “being” is usually a verb, but in this context it has been used as a noun.

[ ]:
```
error_parser(next(error_iter))
```

```
Token          pred   gold
==========================
Urethane       NOUN   NOUN
foams          NOUN   NOUN
are            VERB   VERB
,              .      .
basically      ADV    ADV
,              .      .
reaction       NOUN   NOUN
products       NOUN   NOUN
of             ADP    ADP
hydroxyl-rich  ADJ    ADJ
materials      NOUN   NOUN
and            CONJ   CONJ
polyisocyanatesNOUN   NOUN
(              .      .
usually        ADV    ADV
tolylene       ADJ    NOUN
diisocyanate   NOUN   NOUN
)              .      .
.              .      .
```

The word that is misclassified is as an ADJ instead og a NOUN is a very rare word. Classifying a seldom word before a noun as a adjective is a safe bet. This time the classifier was wrong.

[ ]:
```
error_parser(next(error_iter))
```

```
Token          pred   gold
```

```
==========================
I               PRON  PRON
had             VERB  VERB
brushed         VERB  VERB
my              DET   DET
teeth           NOUN  NOUN
,               .     .
showered        VERB  VERB
,               .     .
shaved          VERB  VERB
and             CONJ  CONJ
dressed         VERB  VERB
by              ADP   ADP
the             DET   DET
time            NOUN  NOUN
a               DET   DET
waiter          NOUN  NOUN
wheeled         VERB  VERB
in              ADP   PRT
breakfast       NOUN  NOUN
.               .     .
```

The word "in" was misclassified as ADP instead of PRT. The classifier is wrong, but I get why. I would also probably tag this as a ADP.

[ ]:
```
error_parser(next(error_iter))
```

```
Token           pred   gold
==========================
The             DET   DET
poet's          NOUN  NOUN
intentions      NOUN  NOUN
are             VERB  VERB
difficult       ADJ   ADJ
to              PRT   PRT
discern         VERB  VERB
and             CONJ  CONJ
,               .     .
except          ADP   ADP
to              PRT   ADP
biographers     NOUN  NOUN
,               .     .
unimportant     ADJ   ADJ
;               .     .
;               .     .
```

The classifier seems to struggle with the differene between particles and adpositions.

[ ]:
```
error_parser(next(error_iter))
```

```
Token           pred   gold
=========================
This            DET    DET
was             VERB   VERB
the             DET    DET
very            ADV    ADJ
sort            NOUN   NOUN
of              ADP    ADP
legislation     NOUN   NOUN
that            ADP    PRON
Roosevelt       NOUN   NOUN
himself         PRON   PRON
had             VERB   VERB
in              ADP    ADP
mind            NOUN   NOUN
.               .      .
```

The classifier misclassified the word "very" as an adverb. It must have mistaken the context, and interpreted the next word sort. The classifier most likely the next word was the verb to sort.

####**1.8 Ex 8: Final testing (10 points)** ###**1.8.1 Part a.** We have reached a stage where we will make no further adjustments to our tagger. We are ready to perform the final testing. First, test the final tagger from exercise 5 on the the test set test?

How is the result compared to dev_test?

[ ]:
```
print(f"Accuracy: {tagger_scikit_entire_brown.accuracy(test):.3f}")
```

```
0.976
```

The results from the dev set and the test set are pretty much the same.

###**1.8.2 Part b.** We will compare in-domain to out-of-domain testing. 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 compared to hobbies?

[ ]:
```
adventure = [sent for sent in brown.tagged_sents(categories="adventure",␣
 ↪tagset="universal")]
hobbies = [sent for sent in brown.tagged_sents(categories="hobbies",␣
 ↪tagset="universal")]
```

[ ]:
```
print(f"Accuracy: {tagger_scikit_entire_brown.accuracy(adventure):.3f}")
```

```
Accuracy: 0.972
```

[ ]:
```
print(f"Accuracy: {tagger_scikit_entire_brown.accuracy(hobbies):.3f}")
```

```
Accuracy: 0.964
```

The results doesn't differ much. The differences might just be random. But if we were to interpret it, the reason might be hobbies being more different from the categories the model was trained on,

compared to adventure.

###**1.9 Ex 9: Comparing to other taggers (10 points)** ####**1.9.1 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?

[45]:
```
%%time
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(news_train)
```

```
CPU times: user 1.04 s, sys: 25.5 ms, total: 1.07 s
Wall time: 1.05 s
```

[49]:
```
print(f"Accuracy: {news_hmm_tagger.accuracy(news_dev_test):.4f}")
```

```
Accuracy: 0.9078
```

[47]:
```
%%time
news_hmm_tagger_entire_brown = nltk.HiddenMarkovModelTagger.train(train)
```

```
CPU times: user 2.94 s, sys: 28.2 ms, total: 2.96 s
Wall time: 2.95 s
```

[50]:
```
print(f"Accuracy: {news_hmm_tagger.accuracy(test):.4f}")
```

```
Accuracy: 0.8860
```

The accuracy is worse compared to my best tagger, but training is alot faster.

####**1.9.2 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:

%%time

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?

[52]:
```
%%time
per_tagger = nltk.PerceptronTagger(load=False)
per_tagger.train(train)
```

```
CPU times: user 1min 30s, sys: 420 ms, total: 1min 30s
Wall time: 1min 30s
```

[53]:
```
print(f"Accuracy: {per_tagger.accuracy(test):.4f}")
```

```
Accuracy: 0.9786
```

The perceptron is a bit slower to train, atleast compared to the HMM. But the accuracy on the test set is pretty good, and is on pair with my best tagger.