MANDATORY 2

ARQUIER Paul

```
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]

 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.evaluate(test_sents), 4))
```

```
0.7915
```

EXERCICE 1 : Tag set and baseline

Part a

```
tagged_sents2 = brown.tagged_sents(categories='news',
tagset="universal")

size1 = int(len(tagged_sents) * 0.1)
size2 = int(len(tagged_sents) * 0.2)

news_train, news_dev_test, news_test = tagged_sents2[size2:],
tagged_sents2[size1:size2], tagged_sents2[:size1]

tagger1 = ConsecutivePosTagger(news_train)

print(round(tagger1.evaluate(news_dev_test), 4))
```

```
0.8689
```

The result is 0.8689 for the accuracy. It's higher than the result for the full brown tagset used in introduction. We can explain this result because of the size of the tagger here is smaller.

Part b

```
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)

tagger2 = BaselinePosTagger(news_train)
print(round(tagger2.evaluate(news_dev_test), 4))
```

```
0.7582
```

We can see that the accuracy is not the same with the BaselinePosTagger and the ConsecutivePosTagger. The result is lower but still not bad.

EXERCICE 2 : Scikit-learn and tuning

```
import numpy as np
from sklearn.naive_bayes import BernoulliNB
from sklearn.feature_extraction import DictVectorizer


class ScikitConsecutivePosTagger(nltk.TaggerI):
    def __init__(self, train_sents,
                 features=pos_features, clf = BernoulliNB()):
        self.features = features
        self.classifier = clf
        self.dict = DictVectorizer()

        train_features = []
        train_labels = []
        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_features.append(featureset)
                train_labels.append(tag)
                history.append(tag)

        X_train = self.dict.fit_transform(train_features)
        y_train = np.array(train_labels)
        clf.fit(X_train, y_train)
```

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

        X_test = self.dict.transform(test_features)
        tags = self.classifier.predict(X_test)

        return zip(sentence, tags)
```

Part a

```
tagger3 = ScikitConsecutivePosTagger(news_train)
print(round(tagger3.evaluate(news_dev_test), 4))
```

```
0.857
```

The result is not the same as in the Exercice 1 a . The result is very closed.

Part b

```
import pandas as pd

alphas = [1, 0.5, 0.1, 0.01, 0.001, 0.0001]
taggerAccuracies = []
for alpha in alphas:
    tagger4 = 
ScikitConsecutivePosTagger(news_train,features=pos_features, clf = 
BernoulliNB(alpha=alpha))
    taggerAccuracies.append(round(tagger4.evaluate(news_dev_test), 4))
df = pd.DataFrame(taggerAccuracies, index=alphas,columns = 
['accuracy'])
df
```

```
        accuracy
1.0000    0.8570
0.5000    0.8749
0.1000    0.8695
0.0100    0.8683
0.0010    0.8651
0.0001    0.8631
```

We have differents results when we change the alpha. We get the best result using alpha = 0.5.

Part c

```
 def pos_features2(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
```

```
                 "suffix(2)": sentence[i][-2:],
                 "suffix(3)": sentence[i][-3:]}

    features["actual_word"] = sentence[i]
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features


taggerAccuracies = []
for alpha in alphas:
    tagger6 = 
ScikitConsecutivePosTagger(news_train,features=pos_features2, clf = 
BernoulliNB(alpha=alpha))
    taggerAccuracies.append(round(tagger6.evaluate(news_dev_test), 4))
df = pd.DataFrame(taggerAccuracies, index=alphas,columns = 
['accuracy'])
df
```

```
        accuracy
1.0000    0.8874
0.5000    0.9166
0.1000    0.9244
0.0100    0.9303
0.0010    0.9330
0.0001    0.9340
```

The accuracy is the best with alpha = 0.0001 here.

EXERCICE 3 : Logistic Regression

Part a

```
import warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.linear_model import LogisticRegression
warnings.filterwarnings("ignore", category=ConvergenceWarning)

tagger7 = ScikitConsecutivePosTagger(news_train, 
features=pos_features2, clf=LogisticRegression())
print(round(tagger7.evaluate(news_dev_test), 4))
```

```
0.9515
```

The result is better than using the Naïve Bayes classifier.

Part b

```
C_values = [0.01, 0.1, 1.0, 10.0, 100.0,1000.0]
accuracies = []

for c in C_values:
    clf=LogisticRegression(C=c)
    tagger8=ScikitConsecutivePosTagger(news_train,
features=pos_features2, clf=clf)
    accuracies.append(round(tagger8.evaluate(news_dev_test), 4))


df = pd.DataFrame(accuracies, index=C_values,columns = ['accuracy'])
df
```

```
         accuracy
0.01       0.8499
0.10       0.9265
1.00       0.9515
10.00      0.9537
100.00     0.9531
1000.00    0.9540
```

Here we used differents values of C. In this case, the model will often fit the data almost perfectly. The best C value is 1000.

EXERCICE 4 : Features

Part a

```
def pos_features3(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}

    features["actual_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"] = sentence[i+1]
    return features

tagger9 = ScikitConsecutivePosTagger(news_train,
features=pos_features3, clf=LogisticRegression(C=1000))
print(round(tagger9.evaluate(news_dev_test), 4))
```

```
0.9637
```

Part b

```
def pos_features4(sentence, i, history):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}

    features["actual_word"] = sentence[i]

    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]

        if sentence[i-1].isupper() == True:
            features['prev-word-capitalized'] = "UPPER"
        if sentence[i-1].islower()== True:
            features['prev-word-is-lower'] = "LOWER"
        if sentence[i-1].isalpha()==True:
            features['prev-word-is-isalpha'] = "ALPHA"
        if sentence[i-1].isdigit()==True:
            features['prev-word-is-digit'] = "DIGIT"
        if i < len(sentence) - 1:
            features["next-word"] = sentence[i+1]
            if sentence[i+1].isupper()==True:
                features['next-word-capitalized'] = "UPPER"
            if sentence[i+1].islower()==True:
                features['next-word-is-lower'] = "LOWER"
    return features

tagger_ =
ScikitConsecutivePosTagger(news_train,features=pos_features4, clf =
LogisticRegression(C=1000))
print(round(tagger_.evaluate(news_dev_test), 4))
```

```
0.9636
```

Here we try to have the best feature. We add the upper, lower, alpha and digit features. We obtain a accuracy of 0.9636. It's closed to 0.001 to the accuracy we find in part a. So this feature doesn't help to earn a lot of accuracy.

EXERCICE 5 : Training on a larger corpus

Part a

```
import random

categories = [categories for categories in brown.categories() if
categories != 'adventure' and categories != 'hobbies'
              and categories!= 'news']
rest = list(brown.tagged_sents(categories=categories,
```

```
tagset='universal'))
random.seed(2888)
random.shuffle(rest)


size1 = int(len(rest) * 0.1)
size2 = int(len(rest) * 0.2)

rest_train, rest_test, rest_dev_test = rest[size2:], rest[:size1], 
rest[size1:size2]

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

Part b

```
baseline_tagger = BaselinePosTagger(train)
print(round(baseline_tagger.evaluate(rest_test), 4))
```

```
0.8512
```

Part c 15-30 mins

```
tagger_2 = ScikitConsecutivePosTagger(train, 
                                    features=pos_features4, 
                                    clf = LogisticRegression(C=1000))
print(round(tagger_2.evaluate(dev_test), 4))
```

```
0.9667
```

The accurary here is 0.9667. It's quite better than before.

EXERCICE 6 : Evaluation metrics

Part a

```
from nltk.tag import PerceptronTagger
tagger9 = ScikitConsecutivePosTagger(train, 
                                    features=pos_features4, 
                                    clf = LogisticRegression(C=1000))

gold_data = dev_test
print(tagger9.confusion(gold_data))
```

```
----------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call 
last)
~\AppData\Local\Temp/ipykernel_14148/1773989463.py in <module>
      1 gold_data = dev_test
```

```
----> 2 print(tagger9.confusion(gold_data))

AttributeError: 'ScikitConsecutivePosTagger' object has no attribute
'confusion'
```

Part b

```
print(tagger9.evaluate_per_tag(gold_data))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
last)
~\AppData\Local\Temp/ipykernel_14148/964373991.py in <module>
----> 1 print(tagger9.evaluate_per_tag(gold_data))

AttributeError: 'ScikitConsecutivePosTagger' object has no attribute
'evaluate_per_tag'
```

Part c

In this exercice, i had some problems with confusion & evalute_per_tag functions. Maybe it's due to my version. I try to find a solution on internet but i didn't find some helps.

EXERCICE 8 : Final Testing

Part a

```
tagger_5 = ScikitConsecutivePosTagger(train,
                                      features=pos_features4,
                                      clf = LogisticRegression(C=1000))
print(round(tagger_5.evaluate(test), 4))
```

```
0.9665
```

The result is compared to the dev_test accuracy,

Part b

```
adventure = brown.tagged_sents(categories="adventure",
tagset='universal')
hobbies = brown.tagged_sents(categories="hobbies", tagset='universal')

tagger10 = ScikitConsecutivePosTagger(train,
                                      features=pos_features4,
                                      clf = LogisticRegression(C=1000))
print(round(tagger10.evaluate(adventure), 4))
```

```
0.9614
```

```
print(round(tagger10.evaluate(hobbies), 4))
```

```
0.9505
```

Those 2 results are lower than the one using the test data. It can be explain because we work on the tagger before with the first data and here we used two news genres. Comparing to adventure and hobbies accuracy, the adventure is higher than the second one. We can explain the difference between the two by the fact that there is more similarity between the adventure genre and all the others, the words are more similar.

EXERCICE 9 : Comparing to others taggers

Part a

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

```
0.8995
```

```
news_hmm_tagger = nltk.HiddenMarkovModelTagger.train(train)
print(round(news_hmm_tagger.evaluate(test), 4))
```

```
0.9521
```

HMM taggers are less efficient in terms of results but the speed of execution is much higher and allows to perform more tests quickly

Part b

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

print(round(per_tagger.evaluate(news_test), 4))
```

```
0.9652
```

```
per_tagger = nltk.PerceptronTagger(load=False)
per_tagger.train(train)
print(round(per_tagger.evaluate(test), 4))
```

```
0.9787
```

The Perceptron tagger is a little bit slower than the HMM taggers but faster than the tagger used in the others exercices. The results are also higher when you compare them with the HMM taggers.