HW4_A20439949_Soutonglang

November 7, 2023

1 CS 585 - Homework 4

Tania Soutonglang A20439949

[1]:
```
import pycrfsuite

from collections import Counter

from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import precision_score, recall_score
from itertools import chain
```

1.1 Problem 1 - Reading the Data in ConLL Format

- Write a function that reads a .tsv files in the CoNLL format and returns two "list of lists" as output:
  - A list of sequences of tokens, where a single token may be a word or punctuation.
  - A list of sequences of tags, representing token-level annotation. You should see these 3 tags in your data ("B-Disease", "I-Disease", "O")

[2]:
```
def read_tsv_CoNLL(file_path):
    tokens = []
    tags = []

    with open(file_path, 'r', encoding='utf-8') as file:
        current_tokens = []
        current_tags = []

        for line in file:
            line = line.strip()
            if not line:
                if current_tokens and current_tags:
                    tokens.append(current_tokens)
                    tags.append(current_tags)
                current_tokens = []
                current_tags = []
            else:
                parts = line.split('\t')
                if len(parts) == 2:
                    token, tag = parts
                    current_tokens.append(token)
                    current_tags.append(tag)

    return tokens, tags
```

- Apply your function to train.tsv and test.tsv. To show you have read in the data correctly, show the following in your notebook output:
  - The number of sequences in train and test. (You should see 5432 sequences in train and 940 sequences in test.)
  - The tokens and tags of the first sequence in the training dataset.

[3]:
```
train_sents, train_tags = read_tsv_CoNLL("train.tsv")
test_sents, test_tags = read_tsv_CoNLL("test.tsv")

print("training sequences: ", len(train_sents))
print("testing sequences: ", len(test_sents))
```

```
training sequences:  5432
testing sequences:  940
```

[4]:
```
print(train_sents[0])
print(train_tags[0])
```

```
['Identification', 'of', 'APC2', ',', 'a', 'homologue', 'of', 'the',
'adenomatous', 'polyposis', 'coli', 'tumour', 'suppressor', '.']
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-Disease', 'I-Disease', 'I-Disease',
'I-Disease', 'O', 'O']
```

1.2 Problem 2 - Data Discovery

In this problem you will examine the data that you read into memory in the previous problem. Using the training dataset for analysis, show the following in your notebook output: - The count of each of the 3 tags in the training data: “B-Disease”, “I-Disease”, and “O”. Note that the most frequent token is “O”, since most words are not part of a disease mention.

[5]:
```
tag_counts = {}

for sequence in train_tags:
    tag_counts = Counter(sequence)

print(tag_counts)
```

```
Counter({'O': 27, 'I-Disease': 3, 'B-Disease': 2})
```

- The 20 most common words/tokens that appear with the tags “B-Disease” or “I-Disease”. That is, show words that often appear disease mentions. (You may show frequent “B-Disease” and “I-Disease” words separately, or you may combine them into a single list.)

```
[6]: # create a list of words for "B-Disease" or "I-Disease" tags
     disease_words = [word for i,
                      seq in enumerate(train_sents) for j,
                      word in enumerate(seq)
                      if train_tags[i][j] in ["B-Disease", "I-Disease"]]

     word_counts = Counter(disease_words)

     # get the 20 most common words
     word_counts.most_common(20)
```

```
[6]: [('-', 636),
      ('deficiency', 322),
      ('syndrome', 281),
      ('cancer', 269),
      ('disease', 256),
      ('of', 178),
      ('dystrophy', 176),
      ('breast', 151),
      ('ovarian', 132),
      ('X', 122),
      ('and', 120),
      ('DM', 120),
      ('ALD', 114),
      ('DMD', 110),
      ('APC', 100),
      ('disorder', 94),
      ('muscular', 94),
      ('G6PD', 92),
      ('linked', 81),
      ('the', 78)]
```

Review the list of words that commonly appear in disease mentions. Do you see any patterns? (You do not need to answer in writing, but it may be helpful in Problem 3 where you design a feature.)

The most common words in the top 20 relate to cancer, and after that there are a lot of abbreviations.

1.3 Problem 3 - Building Features

In this problem, you will build the features that you will use in your CRF model.

- Write a function that takes two inputs:
  - A sequence of tokens
  - An integer position, pointing to one token in that sequence

  and returns a list of features, represented as a list of strings. At minimum, include these features:

  - The current word/token in lower case
  - The suffix (last 3 characters) of the current word
  - The previous word/token (position i-1) or "BOS" if at the beginning of the sequence
  - The next word/token (position i+1), or "EOS" if at the beginning of the sequence
  - At least one other feature of your choice

this function was primarily taken from the section 19 notebook

[7]:
```
def word2features(sent, i):
    word = sent[i]
    # get features
    features = [
        'word.lower=' + word.lower(),      # lower-case of word
        'word.suffix=' + word[-3:],        # suffix of word
        'word.prefix=' + word[:3].lower(),        # prefix of word
    ]

    # get prev word
    if i > 0:
        word1 = sent[i-1][0]
        features.extend([
            '-1:word.lower=' + word1.lower(),
        ])
    else:
        features.append('BOS')

    # get next word
    if i < len(sent)-1:
        word1 = sent[i+1][0]
        features.extend([
            '+1:word.lower=' + word1.lower(),
        ])
    else:
        features.append('EOS')

    return features


def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]
```

- Apply your function your train and test token sequences (from output of Problem 1).

[8]:
```
X_train = [sent2features(s) for s in train_sents]
y_train = train_tags

X_test = [sent2features(s) for s in test_sents]
y_test = test_tags
```

- To show that you have completed this step, apply your output to the first 3 words in the first sequence of the training set. Your output should look something like this (note the names and order of your features in your notebook do not need to match this output):

[9]:
```
for i in range(3):
    print(X_train[0][i])
```

```
['word.lower=identification', 'word.suffix=ion', 'word.prefix=ide', 'BOS',
'+1:word.lower=o']
['word.lower=of', 'word.suffix=of', 'word.prefix=of', '-1:word.lower=i',
'+1:word.lower=a']
['word.lower=apc2', 'word.suffix=PC2', 'word.prefix=apc', '-1:word.lower=o',
'+1:word.lower=,']
```

1.4 Problem 4 - Training a CRF Model

In this problem, you will train a CRF model and evaluate it using metrics computed over individual tags.

- Using the python-crfsuite library, train a CRF sequential tagging model using feature sequences that you built in the previous step. Using your training data as input.

[10]:
```
# create pycrgsuite.Trainer
trainer = pycrfsuite.Trainer(verbose=False)

for X_seq, y_seq in zip(X_train, y_train):
    trainer.append(X_seq, y_seq)

# set the parameters
trainer.set_params({
    'c1': 1.0,   # coefficient for L1 penalty
    'c2': 1e-3,  # coefficient for L2 penalty
    'max_iterations': 50,  # stop earlier

    # include transitions that are possible, but not observed
    'feature.possible_transitions': False
})

# possible parameters
trainer.params()
```

[10]:
```
['feature.minfreq',
 'feature.possible_states',
 'feature.possible_transitions',
 'c1',
 'c2',
 'max_iterations',
 'num_memories',
 'epsilon',
 'period',
 'delta',
 'linesearch',
 'max_linesearch']
```

[11]:
```python
# train the model
trainer.train('conll2002-esp.crfsuite')
```

- Apply your model to your test dataset to generate predicted tag sequences.

[12]:
```python
tagger = pycrfsuite.Tagger()
tagger.open('conll2002-esp.crfsuite')
```

[12]:
```
<contextlib.closing at 0x1a9d566ec10>
```

[13]:
```python
y_pred = [tagger.tag(sent) for sent in X_test]
```

- For each of the 3 labels ("B-Disease", "I-Disease", and "O") show precision, recall, f1-score. (You may use the sckit-learn function classification_report to complete this step. You may also want to "flatten" both the true and predicted tags into a single list of tags to apply this function.)

taken from the section 19 notebook

[14]:
```python
def bio_classification_report(y_true, y_pred):
    lb = LabelBinarizer()
    y_true_combined = lb.fit_transform(list(chain.from_iterable(y_true)))
    y_pred_combined = lb.transform(list(chain.from_iterable(y_pred)))

    tagset = set(lb.classes_)
    tagset = sorted(tagset, key=lambda tag: tag.split('-', 1)[::-1])
    class_indices = {cls: idx for idx, cls in enumerate(lb.classes_)}

    return classification_report(
        y_true_combined,
        y_pred_combined,
        labels = [class_indices[cls] for cls in tagset],
        target_names = tagset,
    )
```

[15]:
```python
print(bio_classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

   B-Disease       0.83      0.71      0.77       960
   I-Disease       0.81      0.75      0.78      1087
           O       0.98      0.99      0.98     22450

   micro avg       0.97      0.97      0.97     24497
   macro avg       0.87      0.82      0.84     24497
```

```
weighted avg       0.97      0.97      0.97     24497
 samples avg       0.97      0.97      0.97     24497
```

1.5 Problem 5 - Inspecting the Trained Model

In this problem you will examine parameter weights assigned by your model. You can do this by calling "tagger.info().transitions" and "tagger.info().state_features" on your trained model object.

- In your notebook, show parameter weights given to transitions between the 3 tag types ("BDisease", "I-Disease", and "O").

Taken from section 19 notebook

[16]:
```
transitions = tagger.info().transitions

def print_transitions(trans_features):
    for (label_from, label_to), weight in trans_features:
        print("%-6s ->\t %-7s \t%0.5f" % (label_from, label_to, weight))

print("Top likely transitions:")
print_transitions(Counter(transitions).most_common(5))

print("\nTop unlikely transitions:")
print_transitions(Counter(transitions).most_common()[-5:])
```

```
Top likely transitions:
B-Disease ->     I-Disease      4.40949
O      ->        O              3.54461
O      ->        B-Disease      3.52636
I-Disease ->     I-Disease      3.30651
B-Disease ->     O              -2.30715

Top unlikely transitions:
I-Disease ->     I-Disease      3.30651
B-Disease ->     O              -2.30715
I-Disease ->     O              -3.70075
I-Disease ->     B-Disease      -3.97391
B-Disease ->     B-Disease      -4.82345
```

- Refer back to the feature you designed in Problem 3 (the feature "of your choice"). Show the parameter weights assigned to this feature. You may truncate this list if it is very long. (This may happen if you included a word from the sequence in the feature name, so your feature was expanded to become a larger set of features that grows with your vocabulary)

[17]:
```
state_features = tagger.info().state_features

def print_state_features(state_features):
    for (attr, label), weight in state_features:
        print("%0.6f %-6s \t%s" % (weight, label, attr))
```

```python
features = list(tagger.info().state_features.keys())
prefix_feat = {}
for feat, tag in features:
    if 'word.prefix' == feat.split("=")[0]:
        prefix_feat[feat, tag] = state_features[feat, tag]

print("Top positive:")
print_state_features(Counter(prefix_feat).most_common(10))

print("\nTop negative:")
print_state_features(Counter(prefix_feat).most_common()[-10:])
```

```
Top positive:
7.226251 B-Disease      word.prefix=ank
6.340155 B-Disease      word.prefix=obe
4.873239 B-Disease      word.prefix=goi
4.865676 B-Disease      word.prefix=edm
4.616451 B-Disease      word.prefix=pie
4.411290 O              word.prefix=mut
3.878427 B-Disease      word.prefix=ado
3.492262 O              word.prefix=wit
3.342234 O              word.prefix=bio
3.341892 O              word.prefix=cas

Top negative:
-2.478031 O             word.prefix=dea
-2.484928 O             word.prefix=duc
-2.738019 O             word.prefix=hyp
-2.826305 O             word.prefix=ost
-2.861462 O             word.prefix=elb
-3.022407 O             word.prefix=neo
-3.227033 O             word.prefix=pit
-3.270224 O             word.prefix=ane
-3.580120 O             word.prefix=dys
-4.465175 O             word.prefix=tum
```

1.6 Problem 6 - Document Level Performance

Tag-level accuracy is easy to compute, but it is not very easy to understand. In particular, one disease reference may cover both “B-Disease” and “I-Disease” tokens. To give another view of model performance, compute document-level precision and recall on your experiment output. To do this: - Write a function that aggregates token-level tags to a document-level label. For example, convert a tag sequence like (“O”, “B-Disease”, “I-Disease”, “O”, “O”) to a single label y=1. Your function should assign y=1 to a sequence with one or more disease mentions (at least one “BDisease” tag) and y=0 to a sequence with no disease mentions.

```
[18]: def aggregate_to_document_level(tags):
          # check if there's at least one disease tag in the tag sequence
          if "B-Disease" or "I-Disease" in tags:
              return 1  # Document contains disease mention(s)
          else:
              return 0  # Document does not contain disease mentions
```

- Apply your function to both true and predicted document-level labels from your test set. Use the output to compute document level precision and recall of your model. Show your results in your notebook.

```
[19]: # Apply the function to both true and predicted document-level labels
      y_doc = [aggregate_to_document_level(tags) for tags in y_test]
      y_doc_pred = [aggregate_to_document_level(tags) for tags in y_pred]
```

```
[20]: document_precision = precision_score(y_doc, y_doc_pred)
      document_recall = recall_score(y_doc, y_doc_pred)

      print("Document-level Precision:", document_precision)
      print("Document-level Recall:", document_recall)
```

```
Document-level Precision: 1.0
Document-level Recall: 1.0
```

1.7 Problem 7 - State Transitions

The python-crfsuite library allows you to set a Boolean hyper-parameter called "feature.possible_transitions". If this parameter is True, then the model may output tag-to-tag transitions that were never seen in training data. (You do not need to apply this parameter in your code to answer this question)

- What is an example of one tag-to-tag transition that never occurred in the training data?

O to I-Disease and I-Disease to O

- For this particular experiment, do you think it makes sense to set this parameter to True or False? That is, should you allow transitions that never occurred in the training data? Explain your answer briefly.

For this dataset it would be okay to set the parameter to True because it is smaller and does not have as many transition combinations.

```
[ ]: 
```