Lab4-Assignment about Named Entity Recognition and Classification

This notebook describes the assignment of Lab 4 of the text mining course. We assume you have succesfully completed Lab1, Lab2 and Lab3 as well. Especially Lab2 is important for completing this assignment.

Learning goals

- **going from linguistic input format to representing it in a feature space**
- **working with pretrained word embeddings**
- **train a supervised classifier (SVM)**
- **evaluate a supervised classifier (SVM)**
- **learn how to interpret the system output and the evaluation results**
- **be able to propose future improvements based on the observed results**

Credits

This notebook was originally created by Marten Postma and Filip Ilievski and adapted by Piek vossen

[Points: 18] Exercise 1 (NERC): Training and evaluating an SVM using CoNLL-2003

[4 point] a) Load the CoNLL-2003 training data using the *ConllCorpusReader* and create for both *train.txt* and *test.txt*:

```
[2 points]  -a list of dictionaries representing the features for each training ins
tances, e..g,
```
```
[
{'words': 'EU', 'pos': 'NNP'},
{'words': 'rejects', 'pos': 'VBZ'},
...
]
```
```

[2 points] -the NERC labels associated with each training instance, e.g.,
dictionaries, e.g.,
```
```
[
'B-ORG',
'O',
....
]
```
```

In [8]:

```
from nltk.corpus.reader import ConllCorpusReader

# Adapt the path to point to the CONLL2003 folder on your local machine
conll_path = 'CONLL2003/CONLL2003'
conll_train = ConllCorpusReader(conll_path, 'train.txt', ['words', 'pos', 'ignore', 'chu
nk'])
```

In [9]:
```

```
training_features = []
training_gold_labels = []

for token, pos, ne_label in conll_train.iob_words():
    a_dict = {
        'token': token,
        'pos': pos
    }

    training_features.append(a_dict)
    training_gold_labels.append(ne_label)
```

In [10]:

```
# Adapt the path to point to the CONLL2003 folder on your local machine
conll_test = ConllCorpusReader(conll_path, 'test.txt', ['words', 'pos', 'ignore', 'chunk
'])
```

In [11]:

```
test_features = []
test_gold_labels = []

for token, pos, ne_label in conll_test.iob_words():
    a_dict = {
        'token': token,
        'pos': pos
    }

    test_features.append(a_dict)
    test_gold_labels.append(ne_label)
```

[2 points] b) provide descriptive statistics about the training and test data:

- **How many instances are in train and test?**
- **Provide a frequency distribution of the NERC labels, i.e., how many times does each NERC label occur?**
- **Discuss to what extent the training and test data is balanced (equal amount of instances for each NERC label) and to what extent the training and test data differ?**

Tip: you can use the following `Counter` functionality to generate frequency list of a list:

Calculating the instances in training and test sets:

In [13]:

```
train_instances_count = len(training_features)
test_instances_count = len(test_features)

print(f"Number of instances in training data is: {train_instances_count}")
print(f"Number of instances in test data is: {test_instances_count}" )
```

```
Number of instances in training data is: 203621
Number of instances in test data is: 46435
```

Frequency distribution of the NERC labels:

In [14]:

```
from collections import Counter

train_label_counts = Counter(training_gold_labels)
test_label_counts = Counter(test_gold_labels)

print("NERC label frequency in training data:")
for label, count in train_label_counts.most_common():
    print(f"{label}: {count}")

print("\nNERC label frequency in test data:")
```

```
for label, count in test_label_counts.most_common():
    print(f"{label}: {count}")
```

```
NERC label frequency in training data:
O: 169578
B-LOC: 7140
B-PER: 6600
B-ORG: 6321
I-PER: 4528
I-ORG: 3704
B-MISC: 3438
I-LOC: 1157
I-MISC: 1155

NERC label frequency in test data:
O: 38323
B-LOC: 1668
B-ORG: 1661
B-PER: 1617
I-PER: 1156
I-ORG: 835
B-MISC: 702
I-LOC: 257
I-MISC: 216
```

Analyzing training and test data (im)balance:

In [15]:

```
train_total = sum(train_label_counts.values())
test_total = sum(test_label_counts.values())

print("\nPercentage distribution in training data:")
for label, count in train_label_counts.most_common():
    percentage = (count/train_total) * 100
    print(f"{label}: {percentage}%")

print("\nPercentage distribution in test data:")
for label, count in test_label_counts.most_common():
    percentage = (count/ test_total) *100
    print(f"{label}: {percentage}%")
```

```
Percentage distribution in training data:
O: 83.2811939829389%
B-LOC: 3.506514553999833%
B-PER: 3.241315974285 56%
B-ORG: 3.1042967080998523%
I-PER: 2.223739201752275%
I-ORG: 1.8190658134475326%
B-MISC: 1.6884309575142051%
I-LOC: 0.5682125124618777%
I-MISC: 0.5672302954999731%

Percentage distribution in test data:
O: 82.53041886508022%
B-LOC: 3.592118014428771%
B-ORG: 3.5770431786368038%
B-PER: 3.482287067944438%
I-PER: 2.48950145364488%
I-ORG: 1.7982125551846666%
B-MISC: 1.5117906751372887%
I-LOC: 0.5534618283622268%
I-MISC: 0.4651663615807042%
```

The NERC labels in both datasets show imbalance. The "O" label dominates at around 83% in both sets, while entity categories each represent only 1-3.5% of the data. This imbalance reflects natural language structure where most words aren't named entities. When comparing the training and test sets, their distribution follow similar patterns, with only slight variations in specific entity frequencies. For example, B-ORG appers marginally more often in the test set (3.58%) than in the training (3.10%). The distributional similarity is advantageous for evaluation purposes, as it suggests the test set effectively represents the patterns the model encounters during

training. However, the presistent imbalance across both sets presents challenges for classifier performance, particularly for the less frequent entity types.

[2 points] c) Concatenate the train and test features (the list of dictionaries) into one list. Load it using the *DictVectorizer*. Afterwards, split it back to training and test.

Tip: You've concatenated train and test into one list and then you've applied the DictVectorizer. The order of the rows is maintained. You can hence use an index (number of training instances) to split the_array back into train and test. Do NOT use: `from sklearn.model_selection import train_test_split` **here.**

In [16]:

```
from sklearn.feature_extraction import DictVectorizer

vec = DictVectorizer()
```

In [17]:

```
concatenated_features = training_features + test_features
the_array = vec.fit_transform(concatenated_features)
training_vec, test_vec = the_array[:203621], the_array[203621:]
```

[4 points] d) Train the SVM using the train features and labels and evaluate on the test data. Provide a classification report (sklearn.metrics.classification_report). The train (*lin_clf.fit*) might take a while. On my computer, it took 1min 53s, which is acceptable. Training models normally takes much longer. If it takes more than 5 minutes, you can use a subset for training. Describe the results:

- **Which NERC labels does the classifier perform well on? Why do you think this is the case?**
- **Which NERC labels does the classifier perform poorly on? Why do you think this is the case?**

In [18]:

```
from sklearn import svm

lin_clf = svm.LinearSVC()
```

In [59]:

```
# NOTE: DON'T run this cell again if you've already trained the SVM! It may take up to 10 minutes!
lin_clf.fit(training_vec, training_gold_labels)
```

```
c:\Users\kevyn\anaconda3\Lib\site-packages\sklearn\svm\_base.py:1235: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

Out[59]:

▾ LinearSVC [i] [?]

LinearSVC()

The above warning might not be critical if the model's performance is acceptable. It is a hint to check whether increasing `max_iter` **or adjusting your model or data preprocessing could improve convergence and possibly lead to a better model.**

In [ ]:

```
from sklearn.metrics import classification_report

# Obtain predictions from the SVM model
svm_predictions = lin_clf.predict(test_vec)

# Generate the classification report and print it
class_report = classification_report(test_gold_labels, svm_predictions)
```

```
print(class_report)
```

```
              precision    recall  f1-score   support

       B-LOC       0.81      0.77      0.79      1668
      B-MISC       0.78      0.66      0.71       702
       B-ORG       0.79      0.52      0.63      1661
       B-PER       0.87      0.44      0.58      1617
       I-LOC       0.62      0.53      0.57       257
      I-MISC       0.59      0.59      0.59       216
       I-ORG       0.66      0.48      0.55       835
       I-PER       0.33      0.87      0.48      1156
           O       0.99      0.98      0.98     38323

    accuracy                           0.92     46435
   macro avg       0.71      0.65      0.65     46435
weighted avg       0.94      0.92      0.92     46435
```

- **Which NERC labels does the classifier perform well on? Why do you think this is the case?**

The classifier performs best on the "O" lebel with an f1-score of 0.98, which is significantly highter than any other category. This strong performance stems from the abundant training examples- "O" labels make up about 83% of the dataset, as was shows in question 1b. The model also does reasonably well on "B-LOC" (beginning of location entities) with an f1-score of 0.79. Location entities tend to have more consistent contextual patterns in tet, such asappearing after specific prepositions (in, at, from) and often having distinctive capitalization patterns that make them easier to identify compatrf to to other entity types.

- **Which NERC labels does the classifier perform poorly on? Why do you think this is the case?**

The classifier struggles most with "I-PER" (inside of person entities) which has an f1-score of only 0.48, despite high recall. This suggests the model overextends person entities, incorrectly labeling many tokens as continuations of person names. The "B-PER" category shos the opposite problem- high precision but low recall (0.44), indicating the model misses many person name beginnings. Organization entities ("B-ORG" and "I-ORG") also perform poorly, with f1-scores of 0.63 and 0.55 respectively. This entity types likely present challenges because they can be highly variable in form- organizations might include common words or span multiple tokens with complex internal structures that are difficult to capture with the current feature set.

[6 points] e) Train a model that uses the embeddings of these words as inputs. Test again on the same data as in 2d. Generate a classification report and compare the results with the classifier you built in 2d.

In [10]:

```
import gensim

# The Google model should be in the same directory. If not adapt the path accordingly.
gensim_path = 'GoogleNews-vectors-negative300.bin\\GoogleNews-vectors-negative300.bin'

word_embedding_model = gensim.models.KeyedVectors.load_word2vec_format(gensim_path, bina
ry=True)
```

In [ ]:

```
training_vec = []

for token_dict in training_features:
    word = token_dict['token']

    # Is word in the model vocabulary (loaded with the Google word2vec embeddings)?
    if word in word_embedding_model:
        vector = word_embedding_model[word]  # assign embedding vector value to variable
    else:
        vector = [0] * 300  # Create a vector with 300 zeros as word2vec model has 300 d
imensions

    training_vec.append(vector)
```

203621

```
205021
[ 3.73535156e-02 -2.03125000e-01  2.12890625e-01  2.44140625e-01
 -2.85156250e-01 -3.44238281e-02  6.68945312e-02 -1.87500000e-01
 -3.90625000e-02  8.48388672e-03 -2.89062500e-01 -8.34960938e-02
  9.08203125e-02 -2.73437500e-01 -3.92578125e-01 -1.06445312e-01
 -6.59179688e-02 -9.94873047e-03 -5.41992188e-02 -4.17480469e-02
  2.63671875e-01  7.95898438e-02  1.50390625e-01  1.94335938e-01
  2.12890625e-01  9.86328125e-02 -3.35937500e-01  1.58203125e-01
  2.83203125e-01  2.33398438e-01 -1.19140625e-01 -2.30468750e-01
  2.61718750e-01  5.95703125e-02  2.61230469e-02 -3.41796875e-01
 -1.54296875e-01  1.37695312e-01  9.86328125e-02  5.56640625e-02
  3.14453125e-01  9.81445312e-02  1.58203125e-01  1.97265625e-01
  2.27050781e-02 -7.61718750e-02 -2.96875000e-01  2.18750000e-01
 -3.59375000e-01  1.88476562e-01 -1.08398438e-01  3.15856934e-03
 -5.83496094e-02  1.96289062e-01  1.28906250e-01 -2.31445312e-01
 -3.92578125e-01  1.36108398e-02 -2.94921875e-01 -7.76367188e-02
 -1.85546875e-01 -2.98828125e-01  1.40991211e-02  2.17285156e-02
  1.29882812e-01 -1.80664062e-01 -1.56250000e-02  1.18164062e-01
 -2.67578125e-01 -1.62109375e-01 -1.20605469e-01  2.14843750e-01
  1.88476562e-01  1.36718750e-01 -2.98828125e-01 -7.12890625e-02
  2.12890625e-01  1.83593750e-01  2.28271484e-02  3.49609375e-01
 -3.82812500e-01 -4.16015625e-01  3.14941406e-02  6.98242188e-02
  7.91015625e-02  1.93359375e-01 -5.05371094e-02 -3.00781250e-01
  1.40625000e-01  2.69531250e-01 -4.80957031e-02 -2.98828125e-01
 -2.59765625e-01  1.54296875e-01 -7.66601562e-02 -2.02148438e-01
 -5.49316406e-02 -3.57421875e-01  4.21875000e-01 -1.05957031e-01
 -5.78613281e-02 -4.02832031e-02 -1.35742188e-01  6.22558594e-02
  7.51953125e-02  1.91406250e-01 -1.43554688e-01 -2.00195312e-01
  1.55273438e-01 -2.46093750e-01  2.09960938e-01 -1.63085938e-01
  1.42578125e-01  3.16406250e-01  2.35351562e-01  1.98242188e-01
 -1.35742188e-01  3.61328125e-02  2.98828125e-01  2.07031250e-01
  7.76367188e-02 -4.27246094e-02 -2.46093750e-01 -1.71875000e-01
  4.51660156e-02 -2.42187500e-01  3.97949219e-02 -1.71661377e-03
 -5.39062500e-01 -2.73437500e-02  1.44531250e-01  2.00195312e-01
 -1.85546875e-01  5.95703125e-02 -2.11914062e-01 -2.26562500e-01
 -5.00488281e-02 -2.23632812e-01  2.85156250e-01 -3.06640625e-01
  2.26562500e-01 -4.85839844e-02 -2.05078125e-01  1.02050781e-01
 -1.61132812e-02 -1.33789062e-01  2.67578125e-01  1.06933594e-01
 -2.91015625e-01 -1.19140625e-01  1.52343750e-01  1.79443359e-02
 -4.95605469e-02 -2.08007812e-01  3.55468750e-01 -1.82617188e-01
 -9.66796875e-02 -4.02832031e-02  1.16699219e-01  3.83300781e-02
 -2.42187500e-01 -1.11816406e-01 -9.58251953e-03 -4.12109375e-01
  1.66015625e-01 -1.63085938e-01  1.02539062e-01  6.93359375e-02
 -7.95898438e-02  8.10546875e-02 -1.87500000e-01 -1.38671875e-01
  8.78906250e-02  8.59375000e-02 -1.20605469e-01 -4.02343750e-01
  3.90625000e-02 -2.08984375e-01  1.02050781e-01 -1.07421875e-01
  4.61425781e-02  1.69677734e-02  3.47656250e-01 -6.68945312e-02
  2.09960938e-01  1.98242188e-01  8.54492188e-02 -8.15429688e-02
 -1.47460938e-01 -9.13085938e-02 -1.54296875e-01 -2.61718750e-01
 -2.28271484e-02 -8.30078125e-02  3.26171875e-01  3.73046875e-01
  3.41796875e-02 -3.90625000e-01 -6.39648438e-02 -3.41796875e-01
 -1.19628906e-01 -1.08886719e-01 -5.73730469e-02  2.28515625e-01
 -3.20434570e-04  1.31835938e-01  2.73437500e-01 -2.92968750e-01
 -5.81054688e-02 -2.18750000e-01 -5.12695312e-02 -1.14257812e-01
 -9.66796875e-02  5.00000000e-01 -1.44531250e-01 -3.61328125e-02
  2.33459473e-03 -4.61425781e-02  2.50000000e-01 -5.81054688e-02
 -3.35937500e-01  2.01416016e-02 -9.61914062e-02  3.08593750e-01
 -1.52343750e-01 -1.77734375e-01  3.96484375e-01  2.23632812e-01
  4.43359375e-01 -1.26953125e-01 -9.52148438e-02  4.60815430e-03
  2.10937500e-01 -1.49414062e-01  2.85644531e-02  1.55273438e-01
  5.02929688e-02  1.70898438e-01 -1.84326172e-02 -2.71484375e-01
 -1.77001953e-02  1.29882812e-01  1.20605469e-01  5.29785156e-02
  2.53906250e-01  3.51562500e-02 -3.12500000e-01 -2.17773438e-01
  1.07421875e-02 -1.92382812e-01  2.94189453e-02  4.98046875e-02
 -6.64062500e-02  3.80859375e-01  4.33349609e-03 -7.12890625e-02
  4.16015625e-01 -1.49414062e-01  3.16406250e-01 -1.05957031e-01
 -3.84521484e-03  1.42578125e-01 -5.49316406e-02  1.84570312e-01
  2.37304688e-01 -1.44531250e-01 -3.04687500e-01 -3.78906250e-01
 -2.85156250e-01 -1.75781250e-01 -2.40478516e-02 -5.59082031e-02
  1.66992188e-01  5.27343750e-02 -2.65625000e-01  9.86328125e-02
  1.12792969e-01  1.72119141e-02 -1.25000000e-01  1.46484375e-01
 -1.19628906e-01  1.95312500e-01 -2.88085938e-02  2.61718750e-01
  4.61425781e-02  1.03515625e-01 -1.31835938e-01  3.10546875e-01
```

```
  9.96093750e-02  2.55859375e-01 -2.99072266e-03  1.63574219e-02
 -2.45361328e-02  6.78710938e-02 -1.41601562e-01  4.41406250e-01
  8.44726562e-02  1.57226562e-01  2.59765625e-01 -5.88378906e-02]
```

In [10]:

```
# NOTE: DON'T run this cell again if you've already trained the SVM! It may take up to 10 minutes!
lin_clf.fit(training_vec, training_gold_labels)
```

Out[10]:

▾ LinearSVC [i] [?]

LinearSVC()

Using embeddings to obtain vectors greatly improved the SVM model training time. Without embeddings, the model training time was 8 minutes and 13 seconds, whereas with embeddings, training the SVM model took only 1 minute and 7 seconds.

In [14]:

```
test_vec = []

for token_dict in test_features:
    word = token_dict['token']

    # Is word in the model vocabulary (loaded with the Google word2vec embeddings)?
    if word in word_embedding_model:
        vector = word_embedding_model[word]  # assign embedding vector value to variable
    else:
        vector = [0] * 300  # Create a vector with 300 zeros as word2vec model has 300 dimensions

    test_vec.append(vector)
```

In [16]:

```
from sklearn.metrics import classification_report

# Obtain predictions from the SVM model
svm_predictions = lin_clf.predict(test_vec)

# Generate the classification report and print it
class_report = classification_report(test_gold_labels, svm_predictions)
print(class_report)
```

```
              precision    recall  f1-score   support

       B-LOC       0.76      0.80      0.78      1668
      B-MISC       0.72      0.70      0.71       702
       B-ORG       0.69      0.64      0.66      1661
       B-PER       0.75      0.67      0.71      1617
       I-LOC       0.51      0.42      0.46       257
      I-MISC       0.60      0.54      0.57       216
       I-ORG       0.48      0.33      0.39       835
       I-PER       0.59      0.50      0.54      1156
           O       0.97      0.99      0.98     38323

    accuracy                           0.93     46435
   macro avg       0.68      0.62      0.64     46435
weighted avg       0.92      0.93      0.92     46435
```

Looking at the classification reports from both models, we can analyze how using word embeddings impacted the SVM classifier's performance on NERC.

The original model and the word embedding model show interesting differences. For the "O" label, both models perform excellently with the same f1-score of 0.98, which is expected given the abundance of this class in the

data.

For location entities, the original model slightly outperforms the word embedding model on "B-LOC" (0.79 vs. 0.78), but the embedding model has better recall (0.80 vs. 0.77). This suggests the embedding model identifies more location entities but with slightly lower precision.

The embedding model significantly improves performance on "B-ORG" with an f1-score of 0.66 compared to 0.63 in the original model. The reall for "B-ORG" increased from 0.52 to 0.64, indicating the word embeddings capture semantic context that helps identify organization names more reliably.

For person entities, the embedding model acheives better balance between precision and recall. The otiginal model had high precision (0.87) but poor recall (0.44) for "B-PER", whereas the embedding model reaches a more balanced 0.75 precision and 0.67 recall, yeilding a much improved f1-score (0.71 vs. 0.58)

However, the embedding model underperforms on inside tokens, particularly "I-ORG" (f1-score dropping from 0.55 to 0.39) and "I-LOC" (0.57 to 0.46). This suggests word embeddings may not effectively capture the continuation pattrns of multi-token entities.

These differences likely stem from how each approach represents features. Word embeddings capture semantic similarities between words, helping identify entities based on their meaning and context rather than just surface patterns. This benefits categories like organizations and persons that have more semantic variety. However, the positional information (i.e., whether a token is inside an entity) seems better captures by the explicit, sparse features from the original approach. Overall, despite slightly lower weighted average metrics, the embedding model provides more balances performance across most entity types and trains significantly faster, making it potentially more practical for real-world applications.

[Points: 10] Exercise 2 (NERC): feature inspection using the Annotated Corpus for Named Entity Recognition

[6 points] a. Perform the same steps as in the previous exercise. Make sure you end up for both the training part (*df_train*) and the test part (*df_test*) with:

- **the features representation using DictVectorizer**
- **the NERC labels in a list**

Please note that this is the same setup as in the previous exercise:

- **load both train and test using:**
  - **list of dictionaries for features**
  - **list of NERC labels**
- **combine train and test features in a list and represent them using one hot encoding**
- **train using the training features and NERC labels**

In [1]:

```
import pandas
from sklearn.feature_extraction import DictVectorizer
from sklearn import svm
```

In [2]:

```
##### Adapt the path to point to your local copy of NERC_datasets
path = r"C:\Users\Sandy\Documents\GitHub\Text-Mining-Assignments\lab4\ner_v2.csv"
kaggle_dataset = pandas.read_csv(path, on_bad_lines='skip') # Changed error_bad_lines=Fal
se to on_bad_lines='skip'
```

In [3]:

```
len(kaggle_dataset)
```

Out[3]:

```
1050795
```

In [4]:

```
df_train = kaggle_dataset[:100000]
df_test = kaggle_dataset[100000:120000]
print(len(df_train), len(df_test))
```

```
100000 20000
```

In [5]:

```
#TODO: Check if we need only token and POS or all features
training_features = []
training_gold_labels = []

for index, instance in df_train.iterrows():
   token = instance["word"] #token
   pos = instance["pos"]
   ne_label = instance["tag"]
   a_dict = {
       'token': token,
       'pos': pos,
   }
   training_features.append(a_dict)
   training_gold_labels.append(ne_label)
```

In [6]:

```
test_features = []
test_gold_labels = []

for index, instance in df_test.iterrows():
    token = instance["word"] #token
    pos = instance["pos"]
    ne_label = instance["tag"]
    a_dict = {
        'token': token,
        'pos': pos,
    }
    test_features.append(a_dict)
    test_gold_labels.append(ne_label)
```

In [7]:

```
vec = DictVectorizer()
the_array = vec.fit_transform(training_features+test_features).toarray()
training_features_array = the_array[:100000]
test_features_array = the_array[100000:120000]
print(training_features_array)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 1. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

In [8]:

```
# SVM model + training
# Maax iterations of 20000 was also tried but it performance did not improve
lin_clf2 = svm.LinearSVC()
lin_clf2.fit(training_features_array, training_gold_labels)
```

```
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\svm\_base.py:1249: Con
vergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

Out[8]:

▾ LinearSVC i ?

```
Linear
SVC()
```

[4 points] b. Train and evaluate the model and provide the classification report:

- **use the SVM to predict NERC labels on the test data**
- **evaluate the performance of the SVM on the test data**

Analyze the performance per NERC label.

In [9]:

```
# Testing
pred = lin_clf2.predict(test_features_array)
```

In [10]:

```
from sklearn.metrics import classification_report
print(classification_report(test_gold_labels, pred))
```

```
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in label
s with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels w
ith no true samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in label
s with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels w
ith no true samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in label
s with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\Sandy\anaconda3\envs\textmining\Lib\site-packages\sklearn\metrics\_classificatio
n.py:1565: UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in labels w
ith no true samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
              precision    recall  f1-score   support

       B-art       0.00      0.00      0.00         4
       B-eve       0.00      0.00      0.00         0
       B-geo       0.80      0.76      0.78       741
       B-gpe       0.96      0.92      0.94       296
       B-nat       1.00      0.50      0.67         8
       B-org       0.64      0.51      0.57       397
       B-per       0.81      0.53      0.64       333
       B-tim       0.91      0.76      0.83       393
       I-art       0.00      0.00      0.00         0
       I-eve       0.00      0.00      0.00         0
       I-geo       0.74      0.50      0.60       156
       I-gpe       1.00      0.50      0.67         2
       I-nat       0.80      1.00      0.89         4
       I-org       0.65      0.44      0.53       321
       I-per       0.42      0.90      0.57       319
       I-tim       0.41      0.08      0.14       108
           O       0.98      0.99      0.99     16918

    accuracy                           0.94     20000
   macro avg       0.60      0.49      0.52     20000
weighted avg       0.95      0.94      0.94     20000
```

Evaluation:

Following suit from the previous exercise, the same questions will be answered here

1. **Which NERC labels does the classifier perform well on? Why do you think this is the case?**

Like in the previous exercise, the "O" label has the highest f1 score as well as having, by far, the largest number of occurrences within the data set. Other labels that performed well were B-gpe (geopolitical), with an f1 score of 0.94, and B-tim (beginning of time/temporal) with an f1-score of 0.83. I-nat has a high f1-score as well (0.89) however there were only 4 occurences in the dataset, so I felt it best to reference B-gpe and B-tim which have a more notable number of occurences within the dataset. "O" is expected to perform exceptionally well due to the volume of non-entity words that are dominating the dataset, therefore the model has a lot of data to pull from and is thereby given many opportunities to correct predict the label. B-tim and B-gpe likely perform well because time and location express can be more easily identifiable within text.

1. **Which NERC labels does the classifier perform poorly on? Why do you think this is the case?**

Aside from labels such as I-art, I-eve, and B-eve, which are all 0 due to their 0 occurrences within the dataset, the labels that performed the most poorly are I-tim (inside of a time entity) with an f-1 score of 0.14, and I-org with 0.53. Though I-org doesn't really fall behind all too much with the f1-scores of the other labels, as many of them sit around the .53-.60 range and have a similar amount of dataset occurences. I-tim I suppose is already at a disadvantage, only having around a third of the number of occurrences the majority of other (min. 3 digit number of occurence) labels have. I-org could likely be more difficult to identify, if there are abbreviations, or if the organization consists of multiple words of varying lengths, making it more difficult to accurately predict. However, even though it sits with one of the lower f1-scores, it isn't that far behin B-org, so I suppose while it is difficult to identify, and we can say that I-org is likely more 'difficult' to predict than B-org, it doesn't fall behind by that much. I-tim may perform really poorly due to it being difficult to recoognize within a long sentence or phrase. The recall is exceptionally low at 0.08, so it is possible that it could have missed many of these instances due to lack of proper contextual understanding as well. Although B-tim has a very good f-1 score, it seems difficult for the model to correctly determine whether the succeeding word is a relevant time expression.

End of this notebook