Group8_lab4

March 2, 2025

1 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 welll. 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

1.1 Credits

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

[1]:
```
# Imports
import gensim
import pandas as pd
import numpy as np
from collections import Counter
from nltk.corpus.reader import ConllCorpusReader
from sklearn.metrics import classification_report
from sklearn.feature_extraction import DictVectorizer
from sklearn import svm
```

1.2 [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 instances, e..
```
```
[
{'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',
....
]
```
```

[ ]:
```
## Adapt the path to point to the CONLL2003 folder on your local machine
train = ConllCorpusReader("./CONLL2003", 'train.txt', ['words', 'pos',␣
↪'ignore', 'chunk'])

training_features = []
training_gold_labels = []

for token, pos, ne_label in train.iob_words():
    # create the features dictionary for the instance
    features_dict = {
        'words': token, 'pos': pos
    }
    # append the features and NE label of the instance
    training_features.append(features_dict)
    training_gold_labels.append(ne_label)

# check
print("Sample Training Features:", training_features[:2])
print("Sample Training Labels:", training_gold_labels[:2])
```

```
Sample Training Features: [{'words': 'EU', 'pos': 'NNP'}, {'words': 'rejects',
'pos': 'VBZ'}]
Sample Training Labels: ['B-ORG', 'O']
```

[3]:
```
### Adapt the path to point to the CONLL2003 folder on your local machine
test = ConllCorpusReader("./CONLL2003", 'test.txt', ['words', 'pos', 'ignore',␣
↪'chunk'])

test_features = []
test_gold_labels = []
for token, pos, ne_label in test.iob_words():
     # create the features dictionary for the instance
     features_dict = {
         'words': token, 'pos': pos
     }
```

```
    # append the features and NE label of the instance
    test_features.append(features_dict)
    test_gold_labels.append(ne_label)

# check
print("Sample Test Features:", test_features[:2])
print("Sample Test Labels:", test_gold_labels[:2])
```

```
Sample Test Features: [{'words': 'SOCCER', 'pos': 'NN'}, {'words': '-', 'pos':
':'}]
Sample Test Labels: ['O', 'O']
```

[**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:

[4]:
```
my_list=[1,2,1,3,2,5]
Counter(my_list)
```

[4]:
```
Counter({1: 2, 2: 2, 3: 1, 5: 1})
```

[5]:
```
# determines the number of instances in the train and test data and print those␣
↪numbers
num_train_instances = len(training_features)
num_test_instances = len(test_features)

print(f"Number of instances in training data: {num_train_instances}")
print(f"Number of instances in test data: {num_test_instances}")
```

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

[6]:
```
# compute the frequency of each label and create a dataframe for nicer␣
↪visualization
train_label_distribution = Counter(training_gold_labels)
test_label_distribution = Counter(test_gold_labels)
train_label_df = pd.DataFrame(train_label_distribution.items(),␣
↪columns=['Label', 'Frequency']).sort_values(by='Frequency', ascending=False)
test_label_df = pd.DataFrame(test_label_distribution.items(), columns=['Label',␣
↪'Frequency']).sort_values(by='Frequency', ascending=False)

print("Training Data - NERC Label Frequency")
print(train_label_df)
print("---")
print("Test Data - NERC Label Frequency")
print(test_label_df)
```

```
Training Data - NERC Label Frequency
    Label  Frequency
1       O     169578
5   B-LOC       7140
3   B-PER       6600
0   B-ORG       6321
4   I-PER       4528
6   I-ORG       3704
2  B-MISC       3438
8   I-LOC       1157
7  I-MISC       1155
---
Test Data - NERC Label Frequency
    Label  Frequency
0       O      38323
1   B-LOC       1668
7   B-ORG       1661
2   B-PER       1617
3   I-PER       1156
8   I-ORG        835
5  B-MISC        702
4   I-LOC        257
6  I-MISC        216
```

[7]:
```
# get the total number of labels
train_total_labels = sum(train_label_distribution.values())
test_total_labels = sum(test_label_distribution.values())

# compute the percentage of each label and create a dataframe for nicer␣
 ↪visualization
train_balance = {label: round((count / train_total_labels) * 100, 2) for label,␣
 ↪count in train_label_distribution.items()}
test_balance = {label: round((count / test_total_labels) * 100, 2) for label,␣
 ↪count in test_label_distribution.items()}
train_balance_df = pd.DataFrame(train_balance.items(), columns=['Label',␣
 ↪'Percentage']).sort_values(by='Percentage', ascending=False)
test_balance_df = pd.DataFrame(test_balance.items(), columns=['Label',␣
 ↪'Percentage']).sort_values(by='Percentage', ascending=False)

# print the label distributions for the train and test data
print("Training Data - Label Distribution (%)")
print(train_balance_df)
print("---")
print("Test Data - Label Distribution (%)")
print(test_balance_df)
```

```
Training Data - Label Distribution (%)
    Label  Percentage
1       O       83.28
5   B-LOC        3.51
3   B-PER        3.24
0   B-ORG        3.10
4   I-PER        2.22
6   I-ORG        1.82
2  B-MISC        1.69
7  I-MISC        0.57
8   I-LOC        0.57
---
Test Data - Label Distribution (%)
    Label  Percentage
0       O       82.53
1   B-LOC        3.59
7   B-ORG        3.58
2   B-PER        3.48
3   I-PER        2.49
8   I-ORG        1.80
5  B-MISC        1.51
4   I-LOC        0.55
6  I-MISC        0.47
```

To what extent is the training and test data balanced (equal amount of instances for each NERC label)? Both the training and test datasets are **highly imbalanced**. In both sets, the "O" label dominates, representing over 80% of the data (83.28% in training and 82.53% in test). The remaining NERC labels are present in much smaller proportions (ranging roughly between 0.47% and 3.59%), indicating that there is a significant disparity in the number of instances per label.

To what extent do the training and test data differ? Although the absolute number of instances differs significantly (203621 in training vs. 46435 in test), the **relative distribution of NERC labels is very similar** between the two datasets. The percentages for each label in the training data closely mirror those in the test data, suggesting that the test set is representative of the training set in terms of label distribution, despite minor variations in percentages.

[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.

```
[ ]: # concatenate the features lists and transform them using DictVectorizer
vec = DictVectorizer()
all_features = training_features + test_features
```

```
the_array = vec.fit_transform(all_features)

# split the features into train and test lists again
train_features_array = the_array[:num_train_instances]
test_features_array = the_array[num_train_instances:]

# check whether split has been done correctly
print("Shape of training features array:", train_features_array.shape)
print("Shape of test features array:", test_features_array.shape)
```

```
Shape of training features array: (203621, 27361)
Shape of test features array: (46435, 27361)
```

[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?

[ ]:
```
# initialize the SVM classifier
lin_clf = svm.LinearSVC(max_iter=10000) # max_iter added to remove error␣
 ↪related to convergence
# 5k iterations didn't work, so we increased it to 10k iterations
```

[10]:
```
# train the model on the training data
lin_clf.fit(train_features_array, training_gold_labels)
```

[10]:
```
LinearSVC(max_iter=10000)
```

[11]:
```
# evaluate the model on the test data
test_predictions = lin_clf.predict(test_features_array)
report = classification_report(test_gold_labels, test_predictions)

print("Classification Report:")
print(report)
```

```
Classification Report:
              precision    recall  f1-score   support

       B-LOC       0.81      0.78      0.79      1668
      B-MISC       0.78      0.66      0.72       702
       B-ORG       0.79      0.52      0.63      1661
       B-PER       0.86      0.44      0.58      1617
       I-LOC       0.62      0.53      0.57       257
      I-MISC       0.57      0.59      0.58       216
       I-ORG       0.70      0.47      0.56       835
       I-PER       0.33      0.87      0.48      1156
```

```
           O       0.98      0.98      0.98     38323

    accuracy                           0.92     46435
   macro avg       0.72      0.65      0.65     46435
weighted avg       0.94      0.92      0.92     46435
```

Labels the Classifier Performs Well On & Reason Why

- **Label "O":** The classifier achieves exceptional performance on the "O" label, with a precision and recall of 0.98 (f1-score = 0.98). This is likely because "O" constitutes the vast majority of tokens in the dataset, providing abundant training examples and making non-entity recognition relatively straightforward.
- **Label "B-LOC":** With a precision of 0.81, recall of 0.78, and an f1-score of 0.79, the classifier performs well on B-LOC. Location entities tend to have distinct contextual features (like accompanying prepositions or specific lexical cues) that make them easier for the model to identify accurately.
- **Label "B-MISC":** With a precision of 0.78, recall of 0.66, and an f1-score of 0.72, we can say that B-MISC is a label on which the classifier performs relatively well, especially when compared to the worse performance on many of the other labels. Miscellaneous entities serve as an umbrella category for more specific entities, such as events, nationalities, products, and works of art, which do not fall under the other labels (LOC, ORG, PER). The classifier performed well on the beginning of miscellaneous entities because their contextual patterns might be more distinct or easier to generalize compared to other entity types, given the diverse entity types that the MISC label encompasses.

Labels the Classifier Performs Poorly On & Reason Why

- **Label "I-PER":** The classifier performs poorly on I-PER, with a notably low precision of 0.33 despite a high recall of 0.87, resulting in an f1-score of 0.48. This suggests that while the model captures many true I-PER tokens, it also generates a high number of false positives. The difficulty in correctly identifying multi-word person names (deciding when a name starts as B-PER and continues as I-PER) likely causes this issue.
- **Label "B-PER":** The B-PER label has a low recall of 0.44 despite a high precision of 0.86 (f1-score = 0.58), indicating that the classifier misses a significant number of person entity beginnings. This may be due to challenges in differentiating person names from other similar tokens (such as, perhaps, organization/company names) or insufficient distinctive features in the training examples.
- **Other Labels (e.g., I-ORG and I-LOC):** Labels such as I-ORG (f1-score = 0.56), I-LOC (f1-score = 0.57), and I-MISC (f1-score = 0.58) also show relatively lower performance. These difficulties might stem from the challenges of consistently segmenting multi-token entities and overlapping contextual cues between different entity types.

[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.**

[12]:
```python
# adapt the path to point to your local copy of the Google embeddings model
word_embedding_model = gensim.models.KeyedVectors.load_word2vec_format('./
 ↪GoogleNews-vectors-negative300.bin', binary=True)
```

[13]:
```python
# Training Data
# get the valid tokens and compute how many valid tokens there are
valid_tokens = [(token, ne_label) for token, pos, ne_label in train.iob_words()␣
 ↪if token and token != 'DOCSTART']
num_tokens = len(valid_tokens)

# based on the number of tokens make arrays to store the word embedding␣
 ↪features and their labels
train_word_embeddings = np.zeros((num_tokens, 300))
train_word_embeddings_labels = np.empty(num_tokens, dtype=object)

# go through the valid tokens
for i, (token, ne_label) in enumerate(valid_tokens):
    # if the token exist in Google's embedding model, use the word embedding
    if token in word_embedding_model:
        train_word_embeddings[i] = word_embedding_model[token]

    # add the label to the label array
    train_word_embeddings_labels[i] = ne_label

# check
print("Sample Training Features:", train_word_embeddings[:2])
print("Sample Training Labels:", train_word_embeddings_labels[:2])
```

```
Sample Training Features: [[ 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]
[ 9.82666016e-03  2.26562500e-01  2.81250000e-01 -3.61328125e-01
 -6.07910156e-02  5.88378906e-02  1.02050781e-01 -3.54003906e-02
  1.47460938e-01  1.28906250e-01 -4.66796875e-01 -1.96533203e-02
 -1.11328125e-01  1.08398438e-01 -9.42382812e-02  2.09960938e-01
  1.37695312e-01 -3.49121094e-02  1.62109375e-01  4.41894531e-02
  5.66406250e-02  1.92382812e-01 -3.26171875e-01  4.37500000e-01
  2.00195312e-01 -4.49218750e-01 -3.24218750e-01  1.73828125e-01
 -4.34570312e-02  1.88476562e-01  1.56250000e-01 -2.08007812e-01
 -6.52343750e-01  2.61718750e-01 -1.73828125e-01 -3.08593750e-01
  7.12890625e-02  2.03125000e-01 -2.02148438e-01 -4.21142578e-03
  1.58203125e-01 -4.14062500e-01  1.78710938e-01 -2.53906250e-01
  4.37011719e-02 -5.50781250e-01 -1.20117188e-01 -6.93359375e-02
  2.40234375e-01 -8.34960938e-02 -1.67968750e-01 -1.76757812e-01
 -1.87500000e-01 -1.95312500e-01 -3.80859375e-02  5.67626953e-03
 -1.42578125e-01 -2.39257812e-01  3.20312500e-01  5.54199219e-02
  1.12304688e-01  3.32031250e-01  1.16699219e-01 -1.50390625e-01
 -2.21679688e-01 -1.11328125e-01  1.19628906e-01 -1.36718750e-01
 -1.63085938e-01  9.66796875e-02  5.62500000e-01 -2.29492188e-01
  4.41894531e-02  5.15625000e-01 -3.84765625e-01 -2.49023438e-01
  1.25976562e-01  5.97656250e-01  3.88671875e-01  4.08203125e-01
  8.10546875e-02 -5.02929688e-02  2.39257812e-01 -7.81250000e-03
  2.69531250e-01 -3.94531250e-01 -1.62109375e-01  3.84765625e-01
  1.54296875e-01 -2.79296875e-01 -9.27734375e-02 -2.61718750e-01
 -8.30078125e-02 -5.63964844e-02  7.78198242e-03  4.96093750e-01
 -5.27343750e-02  1.35742188e-01  4.43359375e-01  2.04101562e-01
  1.20605469e-01  4.98046875e-02 -7.37304688e-02  3.47656250e-01
 -5.54199219e-02 -1.34765625e-01  2.81250000e-01 -1.38671875e-01
 -1.45507812e-01 -2.56347656e-02 -5.10253906e-02 -3.24707031e-02
 -2.55859375e-01  1.94335938e-01  2.37304688e-01  2.73437500e-01
 -3.12500000e-01  2.29492188e-02  1.84570312e-01  1.20605469e-01
  1.85546875e-01  2.37304688e-01  7.47070312e-02  7.76367188e-02
  1.82617188e-01 -2.89062500e-01 -7.81250000e-03 -7.71484375e-02
 -4.61425781e-02 -2.96875000e-01 -9.61914062e-02 -2.35351562e-01
  1.36718750e-01 -5.73730469e-02 -3.33984375e-01 -2.51464844e-02
 -8.34960938e-02  4.85839844e-02  7.51953125e-02  2.53295898e-03
  3.28125000e-01 -3.96484375e-01  2.20703125e-01  1.54296875e-01
  8.88671875e-02  1.40991211e-02 -2.81250000e-01  8.97216797e-03
  1.39648438e-01  2.83203125e-01  2.91015625e-01 -5.78613281e-02
  6.44531250e-02 -2.37304688e-01  1.37695312e-01  1.13525391e-02
  1.74804688e-01  1.21582031e-01 -1.32812500e-01 -5.20019531e-02
  2.90527344e-02 -3.98437500e-01  2.19726562e-01 -4.10156250e-01
  1.55273438e-01  1.17675781e-01  1.45507812e-01 -1.50390625e-01
  2.36328125e-01 -6.10351562e-02  2.47192383e-03  5.02929688e-02
  8.98437500e-02 -6.98242188e-02  2.12890625e-01  1.83593750e-01
  3.17382812e-02  5.44433594e-02 -3.59375000e-01  2.12890625e-01
 -2.49023438e-01 -1.13281250e-01 -8.74023438e-02  1.42578125e-01
 -1.22558594e-01 -2.08740234e-02 -1.26953125e-01  2.96875000e-01
 -6.12792969e-02 -9.91210938e-02  1.60156250e-01  4.17968750e-01
 -4.79125977e-03  2.50000000e-01  9.96093750e-02  5.88378906e-02
  5.68847656e-02 -1.77734375e-01 -3.22265625e-01  8.00781250e-02
  2.57568359e-02 -2.27539062e-01  2.11914062e-01 -8.44726562e-02
 -2.91015625e-01  3.49609375e-01  1.30859375e-01  1.40625000e-01
  1.08642578e-02 -2.83203125e-01  9.66796875e-02  1.59179688e-01
 -3.24707031e-02 -3.39355469e-02 -2.81250000e-01 -1.04980469e-01
 -1.50390625e-01 -5.39550781e-02  1.08886719e-01 -4.12597656e-02
  2.62451172e-03 -1.71875000e-01 -9.61914062e-02  4.19921875e-01
  6.22558594e-02 -1.23535156e-01  1.85546875e-01 -6.64062500e-02
  1.23046875e-01  2.87109375e-01  5.95703125e-02 -5.46875000e-02
  8.34960938e-02  2.20703125e-01  1.37695312e-01 -1.50390625e-01
  2.61718750e-01 -1.62109375e-01  4.43359375e-01  1.62109375e-01
  2.34375000e-01  1.79443359e-02  5.68847656e-02  3.22265625e-01
  7.17773438e-02  9.76562500e-02 -3.39843750e-01  2.29492188e-01
  1.55273438e-01 -1.42578125e-01  9.22851562e-02  1.77734375e-01
 -1.17675781e-01  7.56835938e-02  7.42187500e-02  1.37695312e-01
  1.70898438e-01  5.02929688e-02  4.00390625e-01  2.59765625e-01
 -1.66015625e-01  4.76074219e-02  5.29785156e-02 -6.83593750e-02
  3.10546875e-01  3.96484375e-01 -4.06250000e-01 -1.33789062e-01
 -1.35742188e-01 -9.03320312e-02 -3.02734375e-01  1.25976562e-01
  3.59375000e-01 -2.92968750e-01  1.24023438e-01 -2.43164062e-01
 -2.08007812e-01 -1.97265625e-01 -2.64892578e-02  1.33789062e-01
 -1.62109375e-01  6.10351562e-02 -3.57421875e-01  2.03125000e-01
  3.59375000e-01 -1.52587891e-02  1.04980469e-01  1.41601562e-01
  2.30468750e-01 -2.71484375e-01 -7.51953125e-02 -1.85546875e-01
 -2.83203125e-01  3.18359375e-01 -3.61328125e-01  2.51953125e-01
  7.86132812e-02  3.06396484e-02 -1.03515625e-01  2.03125000e-01]]
Sample Training Labels: ['B-ORG' 'O']
```

[14]:
```python
# Test Data
# get the valid tokens and compute how many valid tokens there are
valid_tokens = [(token, ne_label) for token, pos, ne_label in test.iob_words()␣
 ↪if token and token != 'DOCSTART']
num_tokens = len(valid_tokens)

# based on the number of tokens make arrays to store the word embedding␣
 ↪features and their labels
test_word_embeddings = np.zeros((num_tokens, 300))
test_word_embeddings_labels = np.empty(num_tokens, dtype=object)
```

```python
# go through the valid tokens
for i, (token, ne_label) in enumerate(valid_tokens):
    # if the token exist in Google's embedding model, use the word embedding
    if token in word_embedding_model:
        test_word_embeddings[i] = word_embedding_model[token]

    # add the label to the label array
    test_word_embeddings_labels[i] = ne_label

# check
print("Sample Training Features:", test_word_embeddings[:2])
print("Sample Training Labels:", test_word_embeddings_labels[:2])
```

```
Sample Training Features: [[ 1.26953125e-01  2.60009766e-02  2.69531250e-01
-1.32812500e-01
   5.05371094e-02  8.11767578e-03 -1.33789062e-01 -2.91015625e-01
  -2.53906250e-01  2.81250000e-01 -9.91210938e-02 -5.22460938e-02
  -4.88281250e-01  1.12792969e-01 -2.66113281e-02  2.81250000e-01
   2.67578125e-01  3.37890625e-01 -1.77734375e-01  5.76171875e-02
  -1.06445312e-01  3.06640625e-01  3.33984375e-01 -1.85546875e-01
  -1.47705078e-02  6.15234375e-02  2.65625000e-01  3.04687500e-01
   2.41210938e-01  3.02734375e-01  4.46777344e-02  5.73730469e-02
  -1.57226562e-01 -6.64062500e-01 -1.26953125e-01 -1.42578125e-01
  -5.71289062e-02  1.92260742e-03 -1.06933594e-01  1.81884766e-02
  -3.24218750e-01 -6.99218750e-01  1.32812500e-01  6.86645508e-05
   1.62109375e-01 -3.75000000e-01  2.59765625e-01  5.22460938e-02
   1.68457031e-02  3.98437500e-01 -2.08984375e-01  3.53515625e-01
  -4.54101562e-02  1.57226562e-01 -6.05468750e-02 -1.96289062e-01
  -1.05468750e-01 -1.05957031e-01 -2.06298828e-02  4.27246094e-03
  -4.23828125e-01 -2.57812500e-01 -1.06933594e-01  4.12109375e-01
  -2.49023438e-01 -3.33984375e-01  2.42187500e-01 -2.30468750e-01
   3.08593750e-01  1.44531250e-01  2.35351562e-01 -8.00781250e-02
   3.12500000e-02 -1.81640625e-01  7.12890625e-02  1.93359375e-01
   1.74804688e-01  1.29882812e-01  1.43554688e-01 -9.76562500e-02
  -1.84570312e-01  4.19921875e-02 -1.60156250e-01 -5.81054688e-02
  -2.90527344e-02 -2.38281250e-01 -3.45703125e-01  1.86523438e-01
   7.08007812e-03 -3.08593750e-01 -1.65039062e-01 -8.05664062e-02
   4.25781250e-01 -7.56835938e-02  8.30078125e-02  1.36718750e-01
  -1.06445312e-01 -3.29589844e-02  4.27734375e-01  2.12890625e-01
   3.24707031e-02  4.12109375e-01  5.50781250e-01 -6.59179688e-02
   2.00195312e-01  3.16406250e-01  2.46093750e-01 -2.19726562e-01
  -1.30859375e-01 -1.64794922e-02  3.98437500e-01 -1.62109375e-01
  -1.64062500e-01 -3.37890625e-01  8.25195312e-02  4.88281250e-01
  -2.55859375e-01  1.59179688e-01  1.06445312e-01 -2.71484375e-01
  -2.22656250e-01 -1.08398438e-01  1.42578125e-01 -1.51367188e-01
  -2.15820312e-01  1.17187500e-01 -6.05468750e-01  2.19726562e-01
  -4.83398438e-02 -5.37109375e-02  2.61718750e-01 -5.07812500e-01
  2.47070312e-01 -6.40625000e-01 -1.11083984e-02  8.39843750e-02
 -2.02148438e-01  1.85546875e-01  1.97753906e-02  3.02734375e-01
  1.56250000e-01 -1.40625000e-01  2.65625000e-01 -7.95898438e-02
  1.20605469e-01  1.26953125e-01 -9.13085938e-02  3.78906250e-01
  3.26171875e-01 -2.36328125e-01  3.12500000e-01 -2.69775391e-02
 -1.47460938e-01 -2.55859375e-01 -4.27734375e-01  1.65039062e-01
  1.92871094e-02 -2.39257812e-02 -7.17773438e-02 -4.17968750e-01
  4.66308594e-02  1.47460938e-01  3.10546875e-01 -4.66796875e-01
  1.01562500e-01 -2.10571289e-03  1.48437500e-01 -1.76757812e-01
 -3.49609375e-01  1.73339844e-02  2.53906250e-01  2.23632812e-01
 -6.01562500e-01 -1.72851562e-01  9.08203125e-02  3.24707031e-02
  8.10546875e-02  1.03027344e-01 -3.06396484e-02 -1.66992188e-01
 -1.37695312e-01  1.81640625e-01  4.76562500e-01 -3.58886719e-02
 -2.03125000e-01  2.55859375e-01  1.60156250e-01 -2.89062500e-01
  1.14257812e-01  3.61328125e-02 -8.15429688e-02 -3.69140625e-01
 -2.77343750e-01 -2.27539062e-01 -2.06054688e-01 -1.15234375e-01
  2.77343750e-01  2.61718750e-01 -3.12500000e-01 -3.39355469e-02
 -1.85546875e-01 -2.89062500e-01  1.57226562e-01 -8.72802734e-03
 -2.43164062e-01 -3.71093750e-01  1.59179688e-01 -1.48925781e-02
 -5.66406250e-02 -1.76757812e-01  4.32128906e-02  4.02343750e-01
  2.75390625e-01  2.65625000e-01 -7.81250000e-02 -1.15966797e-02
  1.17675781e-01 -2.57812500e-01 -2.89062500e-01  2.18505859e-02
 -1.68945312e-01 -1.51367188e-01  6.07910156e-02  1.42578125e-01
 -1.14135742e-02  1.65039062e-01  4.49218750e-02 -1.84326172e-02
  1.25000000e-01 -3.28125000e-01  3.26171875e-01  3.30078125e-01
 -2.81250000e-01  6.00585938e-02  1.32812500e-01  7.95898438e-02
  1.00097656e-01 -1.10351562e-01 -1.64062500e-01 -3.08593750e-01
 -6.49414062e-02 -1.24023438e-01  1.85546875e-01  4.30297852e-03
 -1.46484375e-01 -4.41894531e-02 -5.00000000e-01  3.12500000e-01
 -4.10156250e-01  3.06640625e-01  2.98828125e-01  2.30468750e-01
  4.58984375e-01  3.30078125e-01  9.66796875e-02  1.21582031e-01
  3.55468750e-01 -1.60156250e-01 -2.33398438e-01  2.42187500e-01
 -5.27343750e-02 -3.32031250e-02  3.10546875e-01 -8.34960938e-02
 -9.22851562e-02  2.73437500e-01 -5.66406250e-01  4.23828125e-01
 -1.54418945e-02  1.25732422e-02 -2.87109375e-01  3.20312500e-01
 -1.37695312e-01 -1.26953125e-01 -3.73535156e-02  1.30462646e-03
  6.44531250e-02 -1.56250000e-02  3.68652344e-02  1.48437500e-01
 -1.07421875e-01  2.32421875e-01 -2.36328125e-01 -2.40234375e-01
  3.18359375e-01 -5.68847656e-02  2.02148438e-01 -3.88183594e-02
  2.98828125e-01 -1.51367188e-01  5.68847656e-02 -2.47070312e-01
  3.47900391e-03 -6.29882812e-02 -7.56835938e-02  9.96093750e-02
  1.25000000e-01  5.71289062e-02  5.54199219e-02  4.33593750e-01]
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
```

```
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
```

```
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
Sample Training Labels: ['O' 'O']
```

[15]:
```
print("Shape of training features array:", train_word_embeddings.shape)
print("Shape of test features array:", test_word_embeddings.shape)
```

```
Shape of training features array: (203621, 300)
Shape of test features array: (46435, 300)
```

[16]:
```
# initialize the SVM classifier
lin_clf_we = svm.LinearSVC(max_iter=10000)
```

[17]:
```
# train the model on the training data
lin_clf_we.fit(train_word_embeddings, train_word_embeddings_labels)
```

[17]:
```
LinearSVC(max_iter=10000)
```

[18]:
```
# evaluate the model on the test data
test_predictions_we = lin_clf_we.predict(test_word_embeddings)
report_we = classification_report(test_word_embeddings_labels,␣
 ↪test_predictions_we)

print("Classification Report:")
print(report_we)
```

```
Classification 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
```

Comparison of Results With and Without Word Embeddings

- Starting with overall perfermance results, the accuracy was very similar - without word embeddings (1d), it is 0.92, while with word embeddings (1e) it improves to 0.93. The f1-score is slightly higher in the case of not using word embeddings (0.65 compared to 0.64). Therefore, SVM without word embeddings handles minority classes such as I-ORG, I-LOC, I-MISC, B-MISC, and B-LOC slighly better.

- Looking at the results per category, we can observe significant increase when using word embeddings in certain categories. First, B-ORG f1-score increases from 0.63 to 0.66. This means that word embeddings can help to recognize and classify organization names better. Also, B-PER and I-PER results are improved with f1-score (0.58 –> 0.71) and (0.48 –> 0.54), meaning that embeddings are also useful for person name recognition.

  Reasons that could explain the improvements in these categories:

  – semantic representation (meaning and relationships between words);
  – ability to handle variations in entity names (generalize across different forms of the same entity);
  – contextual understanding of named entities (when certain word sequences belong to specific entity types);
  – the traditional SVM without word embeddings relies on handcrafted features that do not recognize deeper connections and relationships between words.

- However, for other categories such as I-LOC, I-ORG, the embeddings lowered the scores. The f1-score of I-LOC is significantly lower when using word embeddings (0.57 to 0.46), meaning that model in 1e has even more difficulties recognizing multi-word location entities. Similarly, for I-ORG, the f1-score descreases from 0.56 to 0.39. Embeddings fail to take into account the contextual dependencies of inside-organization entities.

  Reasons that could explain the deteriotation in these categories:

  – The SVM with word embeddings does not use some details employed by the traditional SVM (e.g. capitalization). It also does not have POS tags as features, which the traditional SVM utilizes. These minor differences may have caused the worse performance in inside-entity recognition;

  - Word embeddings can cause ambiguity by incorrectly grouping similar words together based on their context. The same word can have different meanings and word embeddings treat identical words the same, regardless of the specific context.

- It is also important to note that the O label remains nearly identical in both models (without word embeddings, the precision, recall and f1-score are 0.98, while, with word embeddings, the results are 0.97 (precision), 0.99 (recall), 0.98 (f1-score)). Both models perform equally well in classifying non-entity words. This can be explained by the fact that the O label was the majority of the dataset (see exercise 1b). Both models can perform well on this category because there are many examples of these words, making it easier to do the classfication. Moreover, O label words are less complex (no complex context, semantics), and they are generalizable.

In conclusion, the word embeddings model in 1e performs better for recognizing organizations and person names but is less efficient with classifying inside entity labels for locations and organizations compared to the traditional SVM. The choice between these two depends on the prioritized features and the specific goals of the classification task.

1.3 [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

[19]:
```python
##### Adapt the path to point to your local copy of NERC_datasets
path = './ner_v2.csv'
kaggle_dataset = pd.read_csv(path, on_bad_lines="warn", encoding = 'latin1')
```

```
/var/folders/ld/b7r007xx5qz0wj4jq0413dtm0000gn/T/ipykernel_40025/410003421.py:3:
ParserWarning: Skipping line 281837: expected 25 fields, saw 34

  kaggle_dataset = pd.read_csv(path, on_bad_lines="warn", encoding = 'latin1')
```

[20]:
```python
len(kaggle_dataset)
```

[20]:
```
1050795
```

[21]:
```python
# given code
df_train = kaggle_dataset[:100000]
df_test = kaggle_dataset[100000:120000]

print(len(df_train), len(df_test))
```

```
100000 20000
```

When we include all of the columns of the dataset as features, this causes the vectors to become very sparse, resulting in the classifier being able to only properly predict the `O` class. To avoid this, we have selected a subset of the columns as features of the model, which significantly increased performance.

[22]:
```
selected_features = [
    'word', 'lemma','shape', 'pos',
    'prev-word','prev-pos', 'prev-shape', 'prev-iob', 'prev-prev-iob'
    'next-word','next-pos', 'next-shape',
    'next-next-word', 'next-next-pos','next-next-shape'
    'sentence_id'
]
```

Features Breakdown & Justification

- **`word`** – The current token; core input for entity recognition.
- **`lemma`** – Base form of the word; reduces sparsity by normalizing word variations. Adding the lemmatized versions of `prev-prev-word`, `prev-word`, `next-word`, and `next-next-word`did not improve the performance and were therefore not included in the final selection.
- **`shape`** – Captures capitalization patterns of the word; helps detect named entities, which have distinct capitalization.
- **`pos`** – Part-of-Speech tag; used for distinguishing between entity types (e.g., proper nouns for names).
- **`prev-word`**, **`next-word`**, **`next-next-word`** - The word before, one and two positions after current word. Help capture named entities consisting of multiple words (e.g., ***New*** *York City*, *New* ***York*** *City*, *New York* ***City***).
- **`prev-pos`**, **`next-pos`**, **`next-next-pos`** - POS tags of the word before, one and two positions after the current word. Helps provide context by looking at the surroundings of the current word, as the POS tags of words in a multi-word named entity are often dependent on each other.
- **`prev-shape`**, **`next-shape`**, **`next-next-shape`** – Captures capitalization patterns for names/organizations consisting of more than one word.
- **`prev-iob`** – IOB (Inside-Outside-Beginning) tag of the previous word; helps determine if a word continues an entity (`B-` or `I-`labels).
- **`prev-prev-iob`** – In case of longer named entities, this captures the IOB tag of the word two positions before. Excluding this feature noticably impairs model performance.
- **`sentence_id`** – Identifies which sentence the word belongs to; ensures entity predictions don't span across sentences.

[23]:
```
ner_training_features = []
ner_training_gold_labels = []

for index, instance in df_train.iterrows():
    features_dict = {}
    for key, value in instance.items():
        if key in selected_features:
            features_dict[key] = value
        elif key == "tag":
            ne_label = value

    # append the features and NE label of the instance
    ner_training_features.append(features_dict)
    ner_training_gold_labels.append(ne_label)

# Check
print("Sample Training Feature:", ner_training_features[6])
print("Sample Training Label:", ner_training_gold_labels[6])
```

```
Sample Training Feature: {'lemma': 'london', 'next-next-pos': 'VB', 'next-next-
word': 'protest', 'next-pos': 'TO', 'next-shape': 'lowercase', 'pos': 'NNP',
'prev-iob': 'O', 'prev-pos': 'IN', 'prev-shape': 'lowercase', 'prev-word':
'through', 'shape': 'capitalized', 'word': 'London'}
Sample Training Label: B-geo
```

[24]:
```
ner_test_features = []
ner_test_gold_labels = []

for index, instance in df_test.iterrows():
    features_dict = {}
    for key, value in instance.items():
        if key in selected_features:
            features_dict[key] = value
        elif key == "tag":
            ne_label = value

    # append the features and NE label of the instance
    ner_test_features.append(features_dict)
    ner_test_gold_labels.append(ne_label)

# Check
print("Sample Test Feature:", ner_test_features[3])
print("Sample Test Label:", ner_test_gold_labels[3])
```

```
Sample Test Feature: {'lemma': 'america', 'next-next-pos': 'VBD', 'next-next-
word': 'marched', 'next-pos': '``', 'next-shape': 'punct', 'pos': 'NNP', 'prev-
iob': 'O', 'prev-pos': 'TO', 'prev-shape': 'lowercase', 'prev-word': 'to',
'shape': 'capitalized', 'word': 'America'}
Sample Test Label: B-geo
```

[25]:
```
# checks the number of instances in the train and test data and print those␣
 ↪numbers
```

```
num_ner_train_instances = len(ner_training_features)
num_ner_test_instances = len(ner_test_features)

print(f"Number of instances in training data: {num_ner_train_instances}")
print(f"Number of instances in test data: {num_ner_test_instances}")
```

```
Number of instances in training data: 100000
Number of instances in test data: 20000
```

[26]:
```
# compute the frequency of each label and create a dataframe for nicer␣
 ↪visualization
ner_train_label_distribution = Counter(ner_training_gold_labels)
ner_test_label_distribution = Counter(ner_test_gold_labels)
ner_train_label_df = pd.DataFrame(ner_train_label_distribution.items(),␣
 ↪columns=['Label', 'Frequency']).sort_values(by='Frequency', ascending=False)
ner_test_label_df = pd.DataFrame(ner_test_label_distribution.items(),␣
 ↪columns=['Label', 'Frequency']).sort_values(by='Frequency', ascending=False)

print("Training Data - NERC Label Frequency")
print(ner_train_label_df)
print("---")
print("Test Data - NERC Label Frequency")
print(ner_test_label_df)
```

```
Training Data - NERC Label Frequency
    Label  Frequency
0       O      84725
1   B-geo       3303
5   B-org       1876
10  I-per       1846
7   B-tim       1823
2   B-gpe       1740
3   B-per       1668
6   I-org       1470
4   I-geo        690
12  I-tim        549
8   B-art         75
14  B-eve         53
11  I-gpe         51
15  I-eve         47
9   I-art         43
13  B-nat         30
16  I-nat         11
---
Test Data - NERC Label Frequency
    Label  Frequency
0       O      16918
1   B-geo        741
```

```
4    B-org        397
3    B-tim        393
7    B-per        333
6    I-org        321
8    I-per        319
5    B-gpe        296
2    I-geo        156
9    I-tim        108
10   B-nat          8
12   I-nat          4
13   B-art          4
11   I-gpe          2
```

[27]:
```
ner_train_total_labels = sum(ner_train_label_distribution.values())
ner_test_total_labels = sum(ner_test_label_distribution.values())

# compute the percentage of each label and create a dataframe for nicer␣
 ↪visualization
ner_train_balance = {label: round((count / ner_train_total_labels) * 100, 2)␣
 ↪for label, count in ner_train_label_distribution.items()}
ner_test_balance = {label: round((count / ner_test_total_labels) * 100, 2) for␣
 ↪label, count in ner_test_label_distribution.items()}
ner_train_balance_df = pd.DataFrame(ner_train_balance.items(),␣
 ↪columns=['Label', 'Percentage']).sort_values(by='Percentage',␣
 ↪ascending=False)
ner_test_balance_df = pd.DataFrame(ner_test_balance.items(), columns=['Label',␣
 ↪'Percentage']).sort_values(by='Percentage', ascending=False)

# print the label distributions for the train and test data
print("Training Data - Label Distribution (%)")
print(ner_train_balance_df)
print("---")
print("Test Data - Label Distribution (%)")
print(ner_test_balance_df)
```

```
Training Data - Label Distribution (%)
    Label  Percentage
0       O       84.72
1   B-geo        3.30
5   B-org        1.88
10  I-per        1.85
7   B-tim        1.82
2   B-gpe        1.74
3   B-per        1.67
6   I-org        1.47
4   I-geo        0.69
12  I-tim        0.55
8   B-art        0.07
```

```
11  I-gpe        0.05
14  B-eve        0.05
15  I-eve        0.05
9   I-art        0.04
13  B-nat        0.03
16  I-nat        0.01
---
Test Data - Label Distribution (%)
    Label  Percentage
0       O       84.59
1   B-geo        3.71
4   B-org        1.98
3   B-tim        1.97
7   B-per        1.67
6   I-org        1.60
8   I-per        1.59
5   B-gpe        1.48
2   I-geo        0.78
9   I-tim        0.54
10  B-nat        0.04
12  I-nat        0.02
13  B-art        0.02
11  I-gpe        0.01
```

[28]:
```
# concatenate the features lists and transform them to one hot encoding using␣
 ↪DictVectorizer
vec2 = DictVectorizer()
ner_all_features = ner_training_features + ner_test_features
ner_array = vec2.fit_transform(ner_all_features)  # Joana: adding .toarray()␣
 ↪makes my kernel crash, without it seems to work fine tho

# split the features into train and test lists again
ner_train_features_array = ner_array[:num_ner_train_instances]
ner_test_features_array = ner_array[num_ner_train_instances:]

# check whether split has been done correctly
print("Shape of training features array:", ner_train_features_array.shape)
print("Shape of test features array:", ner_test_features_array.shape)
```

```
Shape of training features array: (100000, 43272)
Shape of test features array: (20000, 43272)
```

[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.

[29]:
```
# initialize the SVM classifier
ner_lin_clf = svm.LinearSVC(max_iter=10000)
```

[30]:
```
# train the model on the training data
ner_lin_clf.fit(ner_train_features_array, ner_training_gold_labels)
```

[30]:
```
LinearSVC(max_iter=10000)
```

[31]:
```
# evaluate the model on the test data
ner_test_predictions = ner_lin_clf.predict(ner_test_features_array)
ner_report = classification_report(ner_test_gold_labels, ner_test_predictions)

print("Classification Report:")
print(ner_report)
```

```
Classification Report:
              precision    recall  f1-score   support

       B-art       1.00      0.50      0.67         4
       B-eve       0.00      0.00      0.00         0
       B-geo       0.86      0.85      0.85       741
       B-gpe       0.90      0.94      0.92       296
       B-nat       1.00      0.75      0.86         8
       B-org       0.74      0.67      0.71       397
       B-per       0.82      0.82      0.82       333
       B-tim       0.95      0.84      0.89       393
       I-geo       0.97      0.96      0.97       156
       I-gpe       1.00      1.00      1.00         2
       I-nat       1.00      1.00      1.00         4
       I-org       0.95      0.93      0.94       321
       I-per       0.94      0.98      0.96       319
       I-tim       0.95      0.89      0.92       108
           O       0.99      0.99      0.99     16918

    accuracy                           0.97     20000
   macro avg       0.87      0.81      0.83     20000
weighted avg       0.97      0.97      0.97     20000


/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Recall
is ill-defined and being set to 0.0 in labels with no true samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Recall
is ill-defined and being set to 0.0 in labels with no true samples. Use
`zero_division` parameter to control this behavior.
```

```
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Recall
is ill-defined and being set to 0.0 in labels with no true samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Analysis of Performance Per Label

- `B-art` - Artistic Work
  - **Precision: 1.00**: every predicted `B-art` entity was correct

  - **Recall: 0.50**: the model missed half of the actual `B-art` entities

  - **F1-score: 0.67**: moderate performance, but the low recall suggests under-detection

  - The very few examples (**Support: 4**) of this category in the test set make it difficult for the model to be evaluated properly (there are also not many examples of this category in the training set)
- `B-eve` - Event
  - **Precision, Recall, F1-score: 0.00**: no instances of this label were present in the test set (**Support: 0**), so the performance of model on this category couldn't be evaluated (there are also few examples of this category in the training set)
- `B-geo` - Geographical Entity
  - **Precision: 0.86**, **Recall: 0.85**, **F1-score: 0.85**: strong and balanced performance across the 3 metrics, meaning the model is relatively accurate in detecting the beginning of location entities
  - **Support: 741**: a well-represented category in the test set, contributing to the good performance
- `B-gpe` - Geopolitical Entity
  - **Precision: 0.90**, **Recall: 0.94**, **F1-score: 0.92**: all metrics are very high, particularly recall, showing that the model is capturing almost all beginnings of geopolitical entities
  - **Support: 296** - sufficient testing data is available
- `B-nat` - Natural Phenomenon
  - **Precision: 1.00**, **Recall: 0.75**, **F1-score: 0.86**: perfect precision, meaning the model only labels something as `B-nat` when it is very sure, but lower recall, suggesting it misses some true entities
  - **Support: 8**, meaning this is a rare category in the test set, affecting proper evaluation of the model on this category (there are also few examples of this category in the training set)
- `B-org` - Organization
  - **Precision: 0.74**, **Recall: 0.67**, **F1-score: 0.71**: both recall and precision are on the lower side, suggesting the model fails to detect the beginning some organization entities and is prone to more false positive classifications

  - **Support: 397**: even though the category is decently represented, the model struggles here
- `B-per` - Person

  - **Precision: 0.82**, **Recall: 0.82**, **F1-score: 0.82**: balanced performance, meaning the model both detects and classifies persons well
  - **Support: 333**: well-represented category in the test set, contributing to good results
- `B-tim` - Time
  - **Precision: 0.95**, **Recall: 0.84**, **F1-score: 0.89**: very high precision indicates the model rarely misclassifies the beginning of time expressions, but slightly lower recall means it can miss some (slightly higher FNs)
  - **Support: 393**: solid performance evaluated on a good amount of test data
- `I-geo` - Inside a Geographical Entity
  - **Precision: 0.97**, **Recall: 0.96**, **F1-score: 0.97**: very high across the 3 metrics, meaning the model correctly classifies multi-word locations
  - **Support: 156**: on the lower side in terms of examples in the test set, but still a strong-performing label
- `I-gpe` - Inside a Geopolitical Entity
  - **Precision, Recall, F1-score: 1.00**: perfect performance, however, since the **Support is only 2** we cannot make a meaningful judgment due to very little test set examples (there are also few examples of this category in the training set)
- `I-nat` - Inside a Natural Phenomenon
  - **Precision, Recall, F1-score: 1.00**: perfect performance, however, since the **Support is only 4** we cannot make a meaningful judgment due to very little test set examples (there are also very few examples of this category in the training set)
- `I-org` - Inside an Organization
  - **Precision: 0.95**, **Recall: 0.93**, **F1-score: 0.94**: very strong performance in recognizing multi-word organizations
  - **Support: 321**: well-represented category in the test set, resulting in trustworthy evaluation of the model
- `I-per` - Inside a Person's Name
  - **Precision: 0.94**, **Recall: 0.98**, **F1-score: 0.96**: high in all metrics, especially recall, suggesting nearly all multi-word names are captured and there are little FPs due to high precision
  - **Support: 319**: good representation in the test set leads to proper evaluation of the performance
- `I-tim` - Inside a Time Expression
  - **Precision: 0.95**, **Recall: 0.89**, **F1-score: 0.92**: slightly lower recall but still suggests strong overall performance
  - **Support: 108**: on the lower side in terms of examples in the test set but still makes model evaluation for this category possible
- `O` - Non-Entity Words
  - **Precision: 0.99**, **Recall: 0.99**, **F1-score: 0.99**: almost perfect for non-entity words, which is expected
  - **Support: 16918**: by far the largest category in the test set, helping drive the overall accuracy of the model up
- `Accuracy`
  - **0.97**: the model correctly predicts `97%` of the tokens in the test set
  - Should be interpreted with caution as it can be misleading if the test set is imbalanced

(i.e., dominated by the `O` label for non-entity words as is the case here)

- `Macro Average` - Average across all entity types, giving equal weight to each class, regardless of frequency
  - **Precision: 0.87, Recall: 0.81, F1-score: 0.83**: the recall (`0.81`) is lower than precision (`0.87`), indicating that the model misses some named entities but it still exhibits fairly high overall performance
- `Weighted Average` - Similar to `Macro Average` but gives more importance to frequent labels
  - **Precision: 0.97, Recall: 0.97, F1-score: 0.97**: since the `O` (non-entities) class dominates, the weighted average is very high, meaning the model performs exceptionally well on the most frequent labels

Conclusion

- **Strongest Categories:** `I-org`, `I-per`, `B-gpe`, `I-geo`, and `I-tim`, show very high metrics and reliabile predictions.
- **Weakest Categories:** `B-eve` (completely missing), `B-nat`, `B-art`, `I-gpe`, `I-nat` cannot be evaluated properly due to low amount of examples in the test set. Moreover, as the dataset is highly imbalanced, these labels are also underrepresented in the training data, making it hard for the model to learn to classify them correctly.

- The model performs well on common entities like locations, persons, and organizations.

- The model handles multi-word entities quite well, as seen with the strong `I-geo`, `I-org`, and `I-per` performance.

1.4 End of this notebook