```
In [1]: from google.colab import drive
        drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
In [2]: !unzip CONLL2003.zip
```

```
Archive:  CONLL2003.zip
   creating: CONLL2003/
  inflating: __MACOSX/._CONLL2003
  inflating: CONLL2003/train.txt
  inflating: __MACOSX/CONLL2003/._train.txt
  inflating: CONLL2003/README.MD
  inflating: __MACOSX/CONLL2003/._README.MD
  inflating: CONLL2003/valid.txt
  inflating: __MACOSX/CONLL2003/._valid.txt
  inflating: CONLL2003/test.txt
  inflating: __MACOSX/CONLL2003/._test.txt
```

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

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 instances, 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 [3]: from nltk.corpus.reader import ConllCorpusReader
        ### Adapt the path to point to the CONLL2003 folder on your local machine
        train = ConllCorpusReader('/content/CONLL2003', 'train.txt', ['words', 'pos', 'ignore', 'chunk'])
        training_features = []
        training_gold_labels = []

        for token, pos, ne_label in train.iob_words():
            a_dict = {
               # add features
                'words': token,
                'pos': pos
            }
            training_features.append(a_dict)
            training_gold_labels.append(ne_label)

        print(training_features[1])
        print(len(training_features))
        print(len(training_gold_labels))
```

```
{'words': 'rejects', 'pos': 'VBZ'}
203621
203621
```

```
In [4]: ### Adapt the path to point to the CONLL2003 folder on your local machine
        train = ConllCorpusReader('/content/CONLL2003', 'test.txt', ['words', 'pos', 'ignore', 'chunk'])
        test_features = []
        test_gold_labels = []

        for token, pos, ne_label in train.iob_words():
            a_dict = {
                # add features
                'words': 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? 203621 in train, 46435 in test.
- Provide a frequency distribution of the NERC labels, i.e., how many times does each NERC label occur? See Counter output below
- 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:

For the training set, the data is not balanced as we can observe that most are labeled as label 'O' and the other entity tokens are between 1155 for 'I-MISC' Miscellaneous entities and 7140 for 'B-LOC' Location entities at the beginning of a chunk of data. For the test set, the data is not balanced as we can observe that most are labeled as label 'O', the rest of the entity tokens are between 216 for 'I-MISC' Miscellaneous entities and 1668 for 'B-LOC' Location entities.

However, ratio-wise, each entity is occurring in a similar percentage in both the training set and the test set which contributes to some extent of balance.

```
In [2]: print(len(training_features))
        print(len(training_gold_labels))
        print(len(test_features))
        print(len(test_gold_labels))
```

```
203621
203621
46435
46435
```

```
In [6]: from collections import Counter

        my_list=[1,2,1,3,2,5]
        Counter(my_list)
        print(Counter(training_gold_labels))
        print(Counter(test_gold_labels))
```

```
Counter({'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})
Counter({'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})
```

[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 [7]: from sklearn.feature_extraction import DictVectorizer
```

```
In [8]: vec = DictVectorizer()
        the_array = training_features + test_features # your code here
        the_array = vec.fit_transform(the_array).toarray()
        the_array.shape

        train_feats = the_array[:len(training_features)]
        test_feats = the_array[len(training_features):]
        print('Number of training words =', train_feats.shape)
        print('Number of test words =', test_feats.shape)
```

```
Number of training words = (203621, 27361)
Number of test words = (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?

The classifier performs especially well on the NERC label 'O' with a recall, precision, and f-1 score of 0.98. The classifier performs quite well on the NERC label 'B-LOC' with a recall of 0.81, precision of 0.78, and f-1 score of 0.79. This is the case due to the fact that these labels had the most data for the training set.

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

The classifier performs poorly on the NERC label 'I-MISC' with a recall of 0.59, precision of 0.57, and f-1 score of 0.58. Additionally, the classifier performs poorly on the NERC label 'I-LOC' with a recall of 0.62, precision of 0.53, and f-1 score of 0.57. This is the case because these labels have the lowest amount of training data for. This lack of training data is effecting negatively the performance of the classifier. Additionally, it is noticeable that the beginning of a chuck (B-...) have higher scores that the following parts of an entity (I-...). More training data for I- labels would probably have a positive effect on the results of the classifier.

```
In [9]: from sklearn import svm
        from sklearn.metrics import classification_report
```

```
In [10]: lin_clf = svm.LinearSVC()
```

```
In [11]: ##### [ YOUR CODE SHOULD GO HERE ]
         lin_clf.fit(train_feats, training_gold_labels)
```

```
Out[11]: ▾ LinearSVC
         LinearSVC()
```

```
In [12]: print(classification_report(test_gold_labels, lin_clf.predict(test_feats)))
```

```
              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
```

[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 [13]: import gensim
         import numpy as np
         import spacy
         ##### Adapt the path to point to your local copy of the Google embeddings model
         word_embedding_model = gensim.models.KeyedVectors.load_word2vec_format('/content/drive/MyDrive/GoogleNews-vectors-negative300.bin.gz', binary=True)
```

```
In [14]: train = ConllCorpusReader('/content/CONLL2003', 'train.txt', ['words', 'pos', 'ignore', 'chunk'])
```

```
In [15]: input_vectors=[]
         labels=[]
         for token, pos, ne_label in train.iob_words():

             if token!='' and token!='DOCSTART':
                 if token in word_embedding_model:
                     vector=word_embedding_model[token]
                 else:
                     vector=[0]*300
                 input_vectors.append(vector)
                 labels.append(ne_label)
```

```
In [16]: lin_clf2 = svm.LinearSVC()
         lin_clf2.fit(input_vectors, labels)
```

```
Out[16]: ▾ LinearSVC
         LinearSVC()
```

```
In [17]: test = ConllCorpusReader('/content/CONLL2003', 'test.txt', ['words', 'pos', 'ignore', 'chunk'])
```

```
In [18]: # your code here
         input_vectors=[]
         labels=[]
         for token, pos, ne_label in test.iob_words():

             if token!='' and token!='DOCSTART':
                 if token in word_embedding_model:
                     vector=word_embedding_model[token]
                 else:
                     vector=[0]*300
                 input_vectors.append(vector)
                 labels.append(ne_label)
```

```
In [19]: print(classification_report(labels, lin_clf2.predict(input_vectors)))
```

```
              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
```

Again, the features with the highest performance are O and B-LOC. However, the classifier that uses the embedded model performs worse on organizations (especially the I labels) now and again the I labels for the locations. Moreover, looking at the overall precision, recall and f1 scores, the classifier using the embeddings model tends to have different scores that vary from entity to entity where they are either a bit higher or a bit lower than the model in 1d. The accuracy of this model heavily depends on the words that are available within the genism word embedding model, as those that are not included will be filled with zero vectors. The more this happens, the worse it will most likely perform for those word entity classes. Finally, looking at the total macro and weighted averages and the total accuracy scores, we see that in general the performance goes down for most scores but not in any significant way, and for accuracy it actually increased by 0.01 (0.92->0.93). For these reasons, we cannot say that is a considerable difference in performance.

[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 [7]: import pandas
```

```
In [8]: with open("/content/ner.csv", encoding="utf-8") as file:
          print(file)
```

```
<_io.TextIOWrapper name='/content/ner.csv' mode='r' encoding='utf-8'>
```

```
In [9]: ##### Adapt the path to point to your local copy of NERC_datasets
        path = "/content/ner.csv"
        kaggle_dataset = pandas.read_csv(path, encoding='cp1252', error_bad_lines=False)
```

```
/usr/local/lib/python3.9/dist-packages/IPython/core/interactiveshell.py:3326: FutureWarning: The error_bad_lines argument has been deprecated and will be removed in a future versio
n.


  exec(code_obj, self.user_global_ns, self.user_ns)
b'Skipping line 281837: expected 25 fields, saw 34\n'
```

```
In [10]: len(kaggle_dataset)
```

```
Out[10]: 1050795
```

```
In [11]: df_train = kaggle_dataset[:100000]
         df_test = kaggle_dataset[100000:120000]
         print(len(df_train), len(df_test))
```

```
100000 20000
```

```
In [12]: kaggle_dataset.columns
```

```
Out[12]: Index(['Unnamed: 0', 'lemma', 'next-lemma', 'next-next-lemma', 'next-next-pos',
       'next-next-shape', 'next-next-word', 'next-pos', 'next-shape',
       'next-word', 'pos', 'prev-iob', 'prev-lemma', 'prev-pos',
       'prev-prev-iob', 'prev-prev-lemma', 'prev-prev-pos', 'prev-prev-shape',
       'prev-prev-word', 'prev-shape', 'prev-word', 'sentence_idx', 'shape',
       'word', 'tag'],
      dtype='object')
```

```
In [52]: for index, instance in kaggle_dataset.iterrows():
             print(index)
             print(instance) # you can access information by using instance['A COLUMN NAME'] which you can use to convert to a dictionary needed for the feature representation.
             break
```

```
0
Unnamed: 0                    0
lemma                  thousand
next-lemma                   of
next-next-lemma        demonstr
next-next-pos               NNS
next-next-shape       lowercase
next-next-word     demonstrators
next-pos                     IN
next-shape            lowercase
next-word                    of
pos                         NNS
prev-iob              __START1__
prev-lemma            __start1__
prev-pos              __START1__
prev-prev-iob         __START2__
prev-prev-lemma       __start2__
prev-prev-pos         __START2__
prev-prev-shape         wildcard
prev-prev-word        __START2__
prev-shape              wildcard
prev-word             __START1__
sentence_idx                1.0
shape               capitalized
word                  Thousands
tag                           O
Name: 0, dtype: object
```

```
In [13]: train_features = []
         train_labels = []

         for index, instance in df_train.iterrows():
             a_dict = {
                # add features
                 'lemma': instance['lemma'],
                 'next-lemma': instance['next-lemma'],
                 'next-next-lemma': instance['next-next-lemma'],
                 'next-next-pos': instance['next-next-pos'],
                 'next-next-shape': instance['next-next-shape'],
                 'next-next-word': instance['next-next-word'],
                 'next-pos': instance['next-pos'],
                 'next-shape': instance['next-shape'],
                 'next-word': instance['next-word'],
                 'pos': instance['pos'],
                 'prev-iob': instance['prev-iob'],
                 'prev-lemma': instance['prev-lemma'],
                 'prev-pos': instance['prev-pos'],
                 'prev-prev-iob': instance['prev-prev-iob'],
                 'prev-prev-lemma': instance['prev-prev-lemma'],
                 'prev-prev-pos': instance['prev-prev-pos'],
                 'prev-prev-shape': instance['prev-prev-shape'],
                 'prev-prev-word': instance['prev-prev-word'],
                 'prev-shape': instance['prev-shape'],
                 'prev-word': instance['prev-word'],
                 'sentence_idx': instance['sentence_idx'],
                 'shape': instance['shape'],
                 'word': instance['word']
                 }
             train_features.append(a_dict)
             train_labels.append(instance['tag'])

         print(train_features[1])
         print(len(train_features))
         print(len(train_labels))
```

```
{'lemma': 'of', 'next-lemma': 'demonstr', 'next-next-lemma': 'have', 'next-next-pos': 'VBP', 'next-next-shape': 'lowercase', 'next-next-word': 'have', 'next-pos': 'NNS', 'next-shap
e': 'lowercase', 'next-word': 'demonstrators', 'pos': 'IN', 'prev-iob': 'O', 'prev-lemma': 'thousand', 'prev-pos': 'NNS', 'prev-prev-iob': '__START1__', 'prev-prev-lemma': '__start
1__', 'prev-prev-pos': '__START1__', 'prev-prev-shape': 'wildcard', 'prev-prev-word': '__START1__', 'prev-shape': 'capitalized', 'prev-word': 'Thousands', 'sentence_idx': 1.0, 'sha
pe': 'lowercase', 'word': 'of'}
100000
100000
```

```
In [14]: test_features = []
         test_labels = []

         for index, instance in df_test.iterrows():
             a_dict = {
                # add features
                 'lemma': instance['lemma'],
                 'next-lemma': instance['next-lemma'],
                 'next-next-lemma': instance['next-next-lemma'],
                 'next-next-pos': instance['next-next-pos'],
                 'next-next-shape': instance['next-next-shape'],
                 'next-next-word': instance['next-next-word'],
                 'next-pos': instance['next-pos'],
                 'next-shape': instance['next-shape'],
                 'next-word': instance['next-word'],
                 'pos': instance['pos'],
                 'prev-iob': instance['prev-iob'],
                 'prev-lemma': instance['prev-lemma'],
                 'prev-pos': instance['prev-pos'],
                 'prev-prev-iob': instance['prev-prev-iob'],
                 'prev-prev-lemma': instance['prev-prev-lemma'],
                 'prev-prev-pos': instance['prev-prev-pos'],
                 'prev-prev-shape': instance['prev-prev-shape'],
                 'prev-prev-word': instance['prev-prev-word'],
                 'prev-shape': instance['prev-shape'],
                 'prev-word': instance['prev-word'],
                 'sentence_idx': instance['sentence_idx'],
                 'shape': instance['shape'],
                 'word': instance['word']
                 }
             test_features.append(a_dict)
             test_labels.append(instance['tag'])

         print(test_features[1])
         print(len(test_features))
         print(len(test_labels))
```

```
{'lemma': 'death', 'next-lemma': 'to', 'next-next-lemma': 'america', 'next-next-pos': 'NNP', 'next-next-shape': 'capitalized', 'next-next-word': 'America', 'next-pos': 'TO', 'next-
shape': 'lowercase', 'next-word': 'to', 'pos': 'NN', 'prev-iob': 'O', 'prev-lemma': '"', 'prev-pos': '``', 'prev-prev-iob': 'O', 'prev-prev-lemma': 'chant', 'prev-prev-pos': 'VBG',
'prev-prev-shape': 'lowercase', 'prev-prev-word': 'chanting', 'prev-shape': 'punct', 'prev-word': '"', 'sentence_idx': 4544.0, 'shape': 'capitalized', 'word': 'Death'}
20000
20000
```

```
In [15]: vec = DictVectorizer()
         the_array = train_features + test_features # your code here
         the_array = vec.fit_transform(the_array).toarray()
         the_array.shape

         train_feats = the_array[:len(train_features)]
         test_feats = the_array[len(train_features):]
         print('Number of training words =', train_feats.shape)
         print('Number of test words =', test_feats.shape)
```

```
Number of training words = (100000, 98137)
Number of test words = (20000, 98137)
```

```
In [16]: lin_clf3 = svm.LinearSVC()
         lin_clf3.fit(train_feats, train_labels)
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

```
Out[16]: ▾ LinearSVC
         LinearSVC()
```

[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 [17]: #Code below gives warning as output because the classifier did not predict some of the true labels based on the test features
         print(classification_report(test_labels, lin_clf3.predict(test_feats)))
```

```
              precision    recall  f1-score   support

       B-art       0.00      0.00      0.00         4
       B-geo       0.88      0.25      0.39       741
       B-gpe       0.26      0.97      0.42       296
       B-nat       0.00      0.00      0.00         8
       B-org       0.37      0.68      0.48       397
       B-per       0.62      0.67      0.64       333
       B-tim       1.00      0.06      0.12       393
       I-geo       0.65      0.90      0.75       156
       I-gpe       0.00      0.00      0.00         2
       I-nat       0.00      0.00      0.00         4
       I-org       0.00      0.00      0.00       321
       I-per       0.70      0.99      0.82       319
       I-tim       0.49      0.44      0.47       108
           O       0.99      0.98      0.98     16918

    accuracy                           0.90     20000
   macro avg       0.43      0.43      0.36     20000
weighted avg       0.93      0.90      0.90     20000
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Precision values range from 0 to 1, with a precision of 1 indicating that there are no false positives. On the other hand, a precision of 0 indicates that the model was incapable of making correct positive predictions. When looking at the precision values of these models, the majority of the classes had low values, such as B-art, B-gpe, B-nat, B-org, I-gpe, I-nat, I-org, I-tim. The classes mentioned had precision values that were below 0.5, thus illustrating that they were incapable of correctly predicting positive instances. However, B-geo, B-per, I-per, I-geo and O had high precision values. Thus, we can assume that this means that the models that had a precision value above 0.5 were capable of predicting positive instances to a certain extent.

Recall values range from 0 to 1, with a recall of 1 indicating that the predictions are true positive. A recall rate of 0 indicates that the predictions were unable to identify any true positives. The recall values are fairly spread, in comparison to the precision rates. B-gpe, I-per, I-geo and O had extremely high recall values, thus the model was able to identify the majority of the positive instances. B-art, B-geo, B-nat, B-tim, I-gpe, I-nat, I-org were unable to identify the positive instances.

F1-scores range from 0 to 1, its a combination of the precision and recall value. Classes I-per, O, I-geo, B-geo, B-per and B-org have fairly high F1-scores meaning that these class models were able to indicate true positive instances. Moreover, it means that these class models had balanced out precision and recall values. For those that had a low F1-score means that the class model performed poorly in indicating the true positive instances. This could be due to having a heavier precision or recall value, causing an imbalance and the need to tune and optimize.

The accuracy displays the accuracy of the model, which was 0.9 out of 1.0. This is a good accuracy rate, however the models performance is not in balance with the classes. The precision, recall and F1-scores have a large range and vary in their values, resulting in different performance levels for each class.

The macro avg is the average for all classes for each column; precision, recall and F1-score. The values were 0.43, 0.43 and 0.36, thus the performance of the model in general for all classes were low. The weighted avg is the weighted average for all classes for each column (precision, recall and F1-score), which is then weighted by the number of instances in each class. The values were 0.93, 0.90 and 0.90, meaning that the performance of the model was in favor for classes that had many samples in their class.

End of this notebook