Lab4-Assignment-nerc

March 12, 2022

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

1.2 [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..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',
....
```

]
```

[109]:
```
from nltk.corpus.reader import ConllCorpusReader
Adapt the path to point to the CONLL2003 folder on your local machine
train = ConllCorpusReader('/Users/lmps/github/ba-text-mining/lab_sessions/lab4/
 ↪CONLL2003', 'train.txt', ['words', 'pos', 'ignore', 'chunk'])
training_features = []
training_gold_labels = []

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

[110]:
```
Adapt the path to point to the CONLL2003 folder on your local machine
test = ConllCorpusReader('/Users/lmps/github/ba-text-mining/lab_sessions/lab4/
 ↪CONLL2003', 'test.txt', ['words', 'pos', 'ignore', 'chunk'])

test_features = []
test_gold_labels = []
for token, pos, ne_label in test.iob_words():
    a_dict = {
        '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? * 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:

[111]:
```
from collections import Counter
import pandas as pd

# How many instances are in train and test?
test_len = len(test.iob_words())
train_len = len(train.iob_words())
print("How many instances are in train and test?\n")
print("Number of instances in the training data is: {}\tPercentage is:␣
 ↪{}\nNumber of instances in the test data is: {}\tPercentage is: {}\n".format(
```
```

```
    train_len, train_len/(test_len + train_len) * 100, test_len, test_len/
 ↪(test_len + train_len) * 100))

# Provide a frequency distribution of the NERC labels
print("Provide a frequency distribution of the NERC labels\n")
train_labels = Counter(training_gold_labels)
test_labels = Counter(test_gold_labels)

print("Test labels are: {}\nTrain labels are: {}\n".format(test_labels.items(),␣
 ↪train_labels.items()))

for key in train_labels.keys():
    print('\nFeature: {}'.format(key))
    print('Training Count is: {}\t\tPercentage is: {}'.
 ↪format(train_labels[key],  train_labels[key]/train_len * 100))
    print('Test Count is: {}\t\tPercentage is: {}'.format(test_labels[key],␣
 ↪test_labels[key]/test_len * 100))

print("\nThe instances for each label is quite similar for both training and␣
 ↪test datasets. For both the test and training dataset, about 13% of the␣
 ↪total instances are labeled LOC, PER, ORG and MISC, where it is almost␣
 ↪equally distributed (~3%) for each label")
```

```
How many instances are in train and test?

Number of instances in the training data is: 203621     Percentage is:
81.43015964423968
Number of instances in the test data is: 46435  Percentage is: 18.56984035576031

Provide a frequency distribution of the NERC labels

Test labels are: dict_items([('O', 38323), ('B-LOC', 1668), ('B-PER', 1617),
('I-PER', 1156), ('I-LOC', 257), ('B-MISC', 702), ('I-MISC', 216), ('B-ORG',
1661), ('I-ORG', 835)])
Train labels are: dict_items([('B-ORG', 6321), ('O', 169578), ('B-MISC', 3438),
('B-PER', 6600), ('I-PER', 4528), ('B-LOC', 7140), ('I-ORG', 3704), ('I-MISC',
1155), ('I-LOC', 1157)])


Feature: B-ORG
Training Count is: 6321         Percentage is: 3.1042967080998523
Test Count is: 1661             Percentage is: 3.5770431786368038

Feature: O
Training Count is: 169578               Percentage is: 83.2811939829389
Test Count is: 38323            Percentage is: 82.53041886508022
```

```
Feature: B-MISC
Training Count is: 3438         Percentage is: 1.6884309575142051
Test Count is: 702              Percentage is: 1.5117906751372887

Feature: B-PER
Training Count is: 6600         Percentage is: 3.24131597428556
Test Count is: 1617             Percentage is: 3.482287067944438

Feature: I-PER
Training Count is: 4528         Percentage is: 2.223739201752275
Test Count is: 1156             Percentage is: 2.48950145364488

Feature: B-LOC
Training Count is: 7140         Percentage is: 3.506514553999833
Test Count is: 1668             Percentage is: 3.592118014428771

Feature: I-ORG
Training Count is: 3704         Percentage is: 1.8190658134475326
Test Count is: 835              Percentage is: 1.7982125551846666

Feature: I-MISC
Training Count is: 1155         Percentage is: 0.5672302954999731
Test Count is: 216              Percentage is: 0.4651663615807042

Feature: I-LOC
Training Count is: 1157         Percentage is: 0.5682125124618777
Test Count is: 257              Percentage is: 0.5534618283622268

The instances for each label is quite similar for both training and test
datasets. For both the test and training dataset, about 13% of the total
instances are labeled LOC, PER, ORG and MISC, where it is almost equally
distributed (~3%) for each label
```

[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.

[112]:
```
from sklearn.feature_extraction import DictVectorizer
import numpy as np
```

[113]:
```
vec = DictVectorizer()
concatenation = training_features + test_features
the_array = vec.fit_transform(concatenation)
```

```
train_array = the_array[:len(training_features)]
test_array = the_array[len(training_features):]
```

[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?

[114]:
```
from sklearn import svm
```

[115]:
```
lin_clf = svm.LinearSVC()
```

[116]:
```
model = lin_clf.fit(train_array, training_gold_labels)
```

[117]:
```
from sklearn.metrics import classification_report

predicted = model.predict(test_array)
print(classification_report(predicted, test_gold_labels))
```

```
              precision    recall  f1-score   support

       B-LOC       0.78      0.81      0.79      1592
      B-MISC       0.66      0.78      0.72       596
       B-ORG       0.52      0.79      0.63      1088
       B-PER       0.44      0.86      0.58       821
       I-LOC       0.53      0.62      0.57       220
      I-MISC       0.59      0.57      0.58       223
       I-ORG       0.47      0.70      0.56       555
       I-PER       0.87      0.33      0.48      3028
           O       0.98      0.98      0.98     38312

    accuracy                           0.92     46435
   macro avg       0.65      0.72      0.65     46435
weighted avg       0.93      0.92      0.92     46435
```

Answer

The O label has the highest performance with a f1-score of 0.98. This can be attributed to the number of instances present (83% of the instances) in the training dataset, since (usually) the greater the representation of a label in the training data, the better the performance. Similarly, B-LOC and B-MISC are the better performing labels, with an f1-score > 0.7. This is surprising as B-MISC has a low representation in the training data (~2% of the instances), however, it achieved a relatively high f1-score. This could be due to the high recall. The NERC label with the lowest f1-score is I-PER even though it has a high number of occurences in the data. However, I-PER does have a high precision score but what impacts it's f1-score is the low recall score it achieved.

Similary, B-PER has a high recall rate but a low precision score which brings down it's f1-score. For other NERC labels with low f1-scores (I-ORG, I-MISC, I-LOC), the classifier performs poorly on both precision and recall. This can be attributed to the their low number of occurences in the dataset.

[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.

[107]:
```
import gensim

word_embedding_model = gensim.models.KeyedVectors.load_word2vec_format("/Users/
↪lmps/github/ba-text-mining/lab_sessions/lab2/GoogleNews-vectors-negative300.
↪bin.gz", binary=True)
```

[118]:
```
def embeddings_conversion(data):
    embedded_data = []
    for word in data:
        # we check if our word
        # is inside the model vocabulary (loaded with the Google word2vec␣
↪embeddings)
        if word in word_embedding_model:
            # in this case the word was found and vector is assigned with its␣
↪embedding vector as the value
            vector=word_embedding_model[word]
        else:
            # if the word does not exist in the embeddings vocabulary,
            # we create a vector with all zeros.
            # The Google word2vec model has 300 dimensions so we creat a vector␣
↪with 300 zeros
            vector=[0]*300
        embedded_data.append(vector)
    return embedded_data

training_embeddings = embeddings_conversion([token for token, pos, ne_label in␣
↪train.iob_words()])
```

[119]:
```
new_lin_cf = svm.LinearSVC()
```

[120]:
```
new_lin_cf.fit(training_embeddings,training_gold_labels)
```

[120]:
```
LinearSVC()
```

[121]:
```
test_embeddings = embeddings_conversion([token for token, pos,ne_label in test.
↪iob_words()])
predicted = new_lin_cf.predict(test_embeddings)
print(classification_report(predicted, test_gold_labels))
```

```
              precision    recall  f1-score   support
```

```
       B-LOC       0.80      0.76      0.78      1760
      B-MISC       0.70      0.72      0.71       674
       B-ORG       0.64      0.69      0.66      1537
       B-PER       0.67      0.75      0.71      1449
       I-LOC       0.42      0.51      0.46       212
      I-MISC       0.54      0.60      0.57       192
       I-ORG       0.33      0.48      0.39       577
       I-PER       0.50      0.59      0.54       988
           O       0.99      0.97      0.98     39046

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

Answer When we compare the classification report of both models, we see some interesting differences. The model that uses the features as an input has a more evenly balanced performance (f1-score) across all the various class labels (low: 0.48, high: 0.98), whereas the model that uses the embeddings has a less balanced performance (low: 0.39, high: 0.98).

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

[101]:
```
import pandas as pd
##### Adapt the path to point to your local copy of NERC_datasets
path = '/Users/lmps/github/ba-text-mining/lab_sessions/lab4/ner_dataset.csv'
kaggle_dataset = pd.read_csv(path, error_bad_lines=False)
```

[102]:
```
len(kaggle_dataset)
```

[102]: 1048575

[103]:
```
df_train = kaggle_dataset[:100000]
df_test = kaggle_dataset[100000:120000]

df_train_labels = df_train['Tag']
df_train_features = []
df_test_labels = df_test['Tag']
df_test_features = []
```

```
for index, entry in df_test.iterrows():
    df_test_features.append({'word': entry.Word, 'pos': entry.POS})

for index, entry in df_train.iterrows():
    df_train_features.append({'word': entry.Word, 'pos': entry.POS})

print(len(df_train_features), len(df_test_features))
```

```
100000 20000
```

[104]:
```
vec = DictVectorizer()
kaggle_concatenation = df_train_features + df_test_features
kaggle_array = vec.fit_transform(kaggle_concatenation)

kaggle_train = kaggle_array[:len(df_train_features)]
kaggle_test = kaggle_array[len(df_train_features):]
```

[105]:
```
kaggle_lin_clf = svm.LinearSVC()
kaggle_model = kaggle_lin_clf.fit(kaggle_train, df_train_labels)
```

[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.

[106]:
```
import warnings
warnings.filterwarnings('ignore')

kaggle_predicted = kaggle_model.predict(kaggle_test)
print(classification_report(kaggle_predicted, df_test_labels))
```

```
              precision    recall  f1-score   support

       B-art       0.00      0.00      0.00         0
       B-eve       0.00      0.00      0.00         2
       B-geo       0.76      0.80      0.78       697
       B-gpe       0.92      0.96      0.94       283
       B-nat       0.50      1.00      0.67         4
       B-org       0.51      0.64      0.57       319
       B-per       0.53      0.81      0.64       220
       B-tim       0.76      0.91      0.83       331
       I-art       0.00      0.00      0.00         4
       I-eve       0.00      0.00      0.00         3
       I-geo       0.50      0.74      0.60       105
       I-gpe       0.50      1.00      0.67         1
       I-nat       1.00      0.80      0.89         5
       I-org       0.44      0.65      0.53       217
       I-per       0.90      0.42      0.57       692
```

```
       I-tim       0.08      0.41      0.14        22
           O       0.99      0.98      0.99     17095

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

Answer

NERC labels B-art, B-eve, B-nat, I-art, I-eve have extremely low f-1 scores of 0. This is most likely because these features were not represented enough in the train data set, moreover, there were only few instances to test them in the test set, thus decreasing the accuracy of the quantative analysis.

NERC labels O, B-gpe, B-tim, and I-nat have the highest f-1 scores. The extremely high score (0.99) of the O label can be attributed to it's very high occurence in the dataset (17095/20000 instances).

Overall, the model has a very high accuracy and weighted average of 0.94. However, it has a low macro average. One explanation for the low macro average score could be that there are many labels that resulted in a f1-score of 0. This thus pulls down the macro average score of the model. Hence, macro average is not a good metric since the performance across classes is not balanced. However, because the instances of such labels are very little (e.g. B-art has 0 and B-eve has 2 support) their f1-score will be penalised in the weighted average score of 0.94. Similarly, the high weighted average score of 0.94 could be explained by the O label, where it has the most instances and 0.99 f1-score. Hence, neither macro nor weighted average can provide a good indication of the performance of our model due to the class and performance imbalance.

1.4 End of this notebook