CHAPTER 8

Named-Entity Recognition Using CRF and BERT

Named-entity recognition (NER) is a natural language processing technique. It is also called *entity identification* or *entity extraction*. It identifies named entities in text and classifies them into predefined categories. For example, extracted entities can be the names of organizations, locations, times, quantities, people, monetary values, and more present in text.

With NER, key information is often extracted to learn what a given text is about, or it is used to gather important information to store in a database.

NER is used in many applications across many domains. NER is extensively used in biomedical data. For instance, it is used for DNA identification, gene identification, and the identification of drug names and disease names. Figure 8-1 shows an example of a medical text-related NER that extracts symptoms, patient type, and dosage.

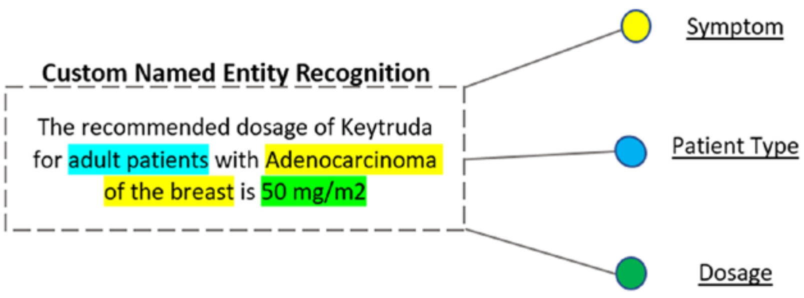

Custom Named Entity Recognition
The recommended dosage of Keytruda for adult patients with Adenocarcinoma of the breast is 50 mg/m2
Symptom
Patient Type
Dosage

Figure 8-1. *NER on medical text data*

NER is also used for optimizing the search queries and ranking search results. It is sometimes combined with topic identification. NER is also used in machine translation.

There are a lot of pretrained general-purpose libraries that use NER. For example, spaCy—an open source Python library for various NLP tasks. And NLTK (natural language tool kit) has a wrapper for the Stanford NER, which is simpler in many cases.

These libraries only extract a certain type of entities, like name, location, and so on. If you need to extract something very domain-specific, such as the name of a medical treatment, it is impossible. You need to build custom NER in those scenarios. This chapter explains how to build a custom NER model.

Problem Statement

The goal is to extract a named entity from movie trivia. For example, the tags are for movie name, actor name, director name, and movie plot. General libraries might extract the names but don't differentiate between actor and director, and it would be challenging to extract movie plots. We need to build the customer model that predicts these tags for the sentences on movies.

Essentially, we have a data set that talks about the movies. It consists of sentences or questions on movies, and each of those words in the sentence has a predefined tag. We need to build the NER model to predict these tags.

Along the way, we need to understand these concepts.

1. Build the model using various algorithms.
2. Design a metric to measure the performance of the model.
3. Understand where the model fails and what might be the reason for the failure.
4. Fine-tune the model.
5. Repeat these steps until we achieve the best accuracy on the test data.

Methodology and Approach

NER identifies the entities in a text and classifies them into predefined categories such as location, a person's name, organization name, and more. But for this problem, we need to tag the director's name, actor's name, genre, and movie character (likewise, there are 25 such tags defined in the data set) for the entities in a sentence. So NER alone does not suffice. Here let's build custom models and train them using conditional random fields and BERT.

The steps to solve this problem are as follows.

1. Data collection
2. Data understanding
3. Data preprocessing
4. Feature mapping
5. Model Building
   - Conditional random fields
   - BERT
6. Hyperparameter tuning
7. Evaluating the model
8. Prediction on random sentences

Figure 8-2 shows how the product works at a high level.

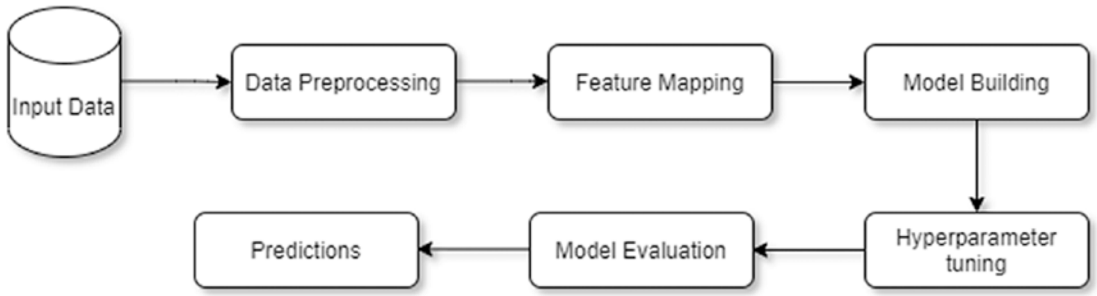


Figure 8-2. *Approach flow*

Implementation

We understood the problem statement as well as various approached to solve it. Let's start the implementation of the project. We begin with importing and understanding the data.

Data

We used data from the MIT movie corpus, which is in .bio format. Download the trivia10k13train.bio and trivia10k13test.bio data sets from https://groups.csail.mit.edu/sls/downloads/movie/.

Now let's convert the data into a pandas data frame using the following code.

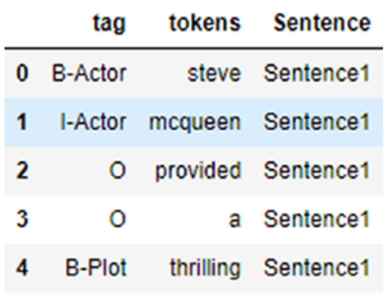
```
#create a function to add a column sentence that indicates the sentence
u=id for each txt file as a preprocessing step.

import pandas as pd
def data_conversion(file_name):
    df_eng=pd.read_csv(file_name,delimiter='\t',header=None,skip_blank_
lines=False)
    df_eng.columns=['tag','tokens']
    tempTokens = list(df_eng['tokens'])
    tempSentence = list()
    count = 1
    for i in tempTokens:
        tempSentence.append("Sentence" + str(count))
        if str(i) == 'nan':
            count = count+1
    dfSentence = pd.DataFrame (tempSentence,columns=['Sentence'])
    result = pd.concat([df_eng, dfSentence], axis=1, join='inner')
    return result

#passing the text files to function
trivia_train=data_conversion('trivia10k13train.txt')
trivia_test=data_conversion('trivia10k13test.txt')
```

Train Data Preparation

Let's look at the first five rows of the training data.

```
trivia_train.head()
```

Figure 8-3 shows the output for the first five rows of the training data.



	tag	tokens	Sentence
0	B-Actor	steve	Sentence1
1	I-Actor	mcqueen	Sentence1
2	O	provided	Sentence1
3	O	a	Sentence1
4	B-Plot	thrilling	Sentence1

Figure 8-3. *Sample training data*

Next, we check the number of rows and columns present in the training data set.

```
trivia_train.shape
```

The following is the output.

```
(166638, 3)
```

There are a total of 166,638 rows and three columns. Let's check how many unique words are present in the training data set.

```
trivia_train.tokens.nunique()
```

The following is the output.

```
10986
```

There are 10,986 unique words and 7816 total sentences for the training data. Now, let's check if there are any null values present in the training data set.

```
trivia_train.isnull().sum()
```

The following is the output.

```
tag         7815
tokens      7816
Sentence       0
dtype: int64
```

There are 7816 null rows. Let's drop the null rows using the following code.

```
trivia_train.dropna(inplace=True)
```

Test Data Preparation

Let's look at the first five rows of the test data.

```
trivia_test.head()
```

Next, we check the number of rows and columns present in the test data set.

```
trivia_test.shape
```

The following is the output.

```
(40987, 3)
```

There are a total of 40,987 rows and three columns. Let's check how many unique words are present in the test data set.

```
trivia_test.tokens.nunique()
```

The following is the output.

```
5786
```

There are 5786 unique words and 1953 total sentences for test data. Now, let's check if there are any null values present in the test data set.

```
trivia_test.isnull().sum()
```

The following is the output.

```
tag         1952
tokens      1952
Sentence       0
dtype: int64
```

There are 1952 null rows. Let's drop the null rows using the following code.

```
trivia_test.dropna(inplace=True)
```

The data set consists of three columns after extraction.

- **tag** is the category of words
- **tokens** consist of words
- **sentence** is the sentence number in which a word belongs

There are a total of 24 (excluding the O tag) unique tags in the given training data set. Their distribution is as follows.

```
#below is the code to get the distribution plot for the tags.

trivia_train[trivia_train["tag"]!="O"]["tag"].value_counts().
plot(kind="bar", figsize=(10,5))
```

Figure 8-4 shows the output of the tag distribution.

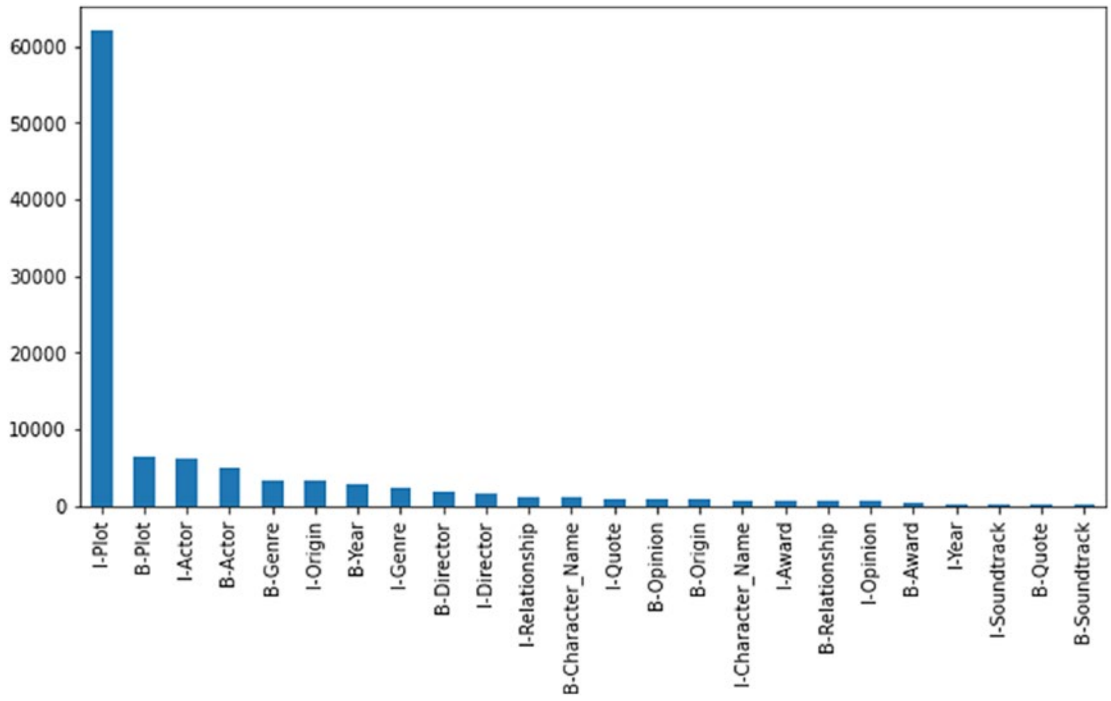


Figure 8-4. *Tag distribution*

Now, let's create a copy of train and test data for the further analysis and model build.

```
data=trivia_train.copy()
data1=trivia_test.copy()
```

Let's rename the columns using the following code.

```
data.rename(columns={"Sentence":"sentence_id","tokens":"words","tag":
"labels"}, inplace =True)
data1.rename(columns={"Sentence":"sentence_id","tokens":"words","tag":
"labels"}, inplace =True)
```

Model Building

Conditional Random Fields (CRF)

CRF is a conditional model class best suited to prediction tasks where the state of neighbors or contextual information affects the current prediction.

The main applications of CRFs are named-entity recognition, part of speech tagging, gene prediction, noise reduction, and object detection problems, to name a few.

In a sequence classification problem, the final goal is to find the probability of *y* (target) given input of sequence vector *x*.

Because conditional random fields are conditional models, they apply logistic regression to sequence data.

Conditional distribution is basically

$$Y = \text{argmax } P(y|x)$$

This finds the best output (probability) given sequence *x*.

In CRF, the input data is expected to be sequential, so we have to label the data as position *i* for the data point we are predicting.

We define feature functions for each variable; in this case, there is no POS tag. We only use one feature function.

The feature function's main purpose is to express a characteristic of the sequence that the data point represents.

Each feature function is relatively based on the label of the previous word and the current word. It is either a 0 or a 1.

So, to build CRF as we do in other models, assign some weights to each feature function and let weights update by optimization algorithm like gradient descent.

Maximum likelihood estimation is used to estimate parameters, where we take the negative log of distribution to make the derivative easier to calculate.

In general,

1. Define feature function.
2. Initialize weights to random values.
3. Apply gradient descent to parameter values to converge.

CRFs are more likely similar to logistic regression since they use *conditional probability distribution*. But, we extend the algorithm by applying feature functions as the sequential input.

We aim to extract entities from the given sentence and identifying their types using the CRF model. Now, let's import the required libraries.

We used the following libraries.

```
#For visualization
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(color_codes=True)
sns.set(font_scale=1)
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

# For Modeling
from sklearn.ensemble import RandomForestClassifier
from sklearn_crfsuite import CRF, scorers, metrics
from sklearn_crfsuite.metrics import flat_classification_report
from sklearn.metrics import classification_report, make_scorer

import scipy.stats
import eli5
```

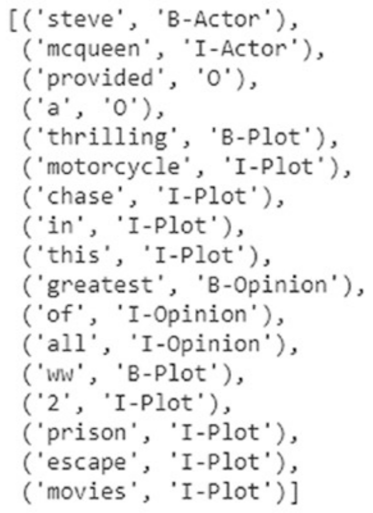
Let's create grouped words and their corresponding tags as a tuple. Also, let's store the words of the same sentence in one list using the following sentence generator function.

```
class Get_Sent(object):

    def __init__(self, dataset):
        self.n_sent = 1
        self.dataset = dataset
        self.empty = False
        agg_func = lambda s: [(a, b) for a,b in zip(s["words"].values.tolist(),
                                            s["labels"].values.tolist())]
        self.grouped = self.dataset.groupby("sentence_id").apply(agg_func)
        self.sentences = [x for x in self.grouped]

    def get_next(self):
        try:
            s = self.grouped["Sentence: {}".format(self.n_sent)]
            self.n_sent += 1
            return s
        except:
            return None
```

```
# calling the Get_Sent function and passing the train dataset
Sent_get= Get_Sent(data)
sentences = Sent_get.sentences
```

So, the sentence looks like this.

Figure 8-5 shows the output of the Get_Sent function for training data.

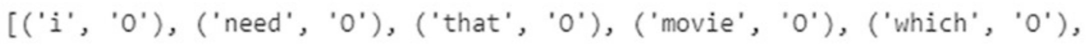


```
[('steve', 'B-Actor'),
 ('mcqueen', 'I-Actor'),
 ('provided', 'O'),
 ('a', 'O'),
 ('thrilling', 'B-Plot'),
 ('motorcycle', 'I-Plot'),
 ('chase', 'I-Plot'),
 ('in', 'I-Plot'),
 ('this', 'I-Plot'),
 ('greatest', 'B-Opinion'),
 ('of', 'I-Opinion'),
 ('all', 'I-Opinion'),
 ('ww', 'B-Plot'),
 ('2', 'I-Plot'),
 ('prison', 'I-Plot'),
 ('escape', 'I-Plot'),
 ('movies', 'I-Plot')]
```

Figure 8-5. *Output*

```
# calling the Get_Sent function and passing the test dataset

Sent_get= Get_Sent(data1)
sentences1 = Sent_get.sentences

#This is what a sentence will look like.
print(sentences1[0])
```

Figure 8-6 shows the output of the Get_Sent function for test data.



```
[('i', 'O'), ('need', 'O'), ('that', 'O'), ('movie', 'O'), ('which', 'O'),
```

Figure 8-6. *Output*

For converting text into numeric arrays, we use simple and complex features.

Simple Feature Mapping

In this mapping, we have considered simple mapping of words and considered only six features of each word.

- word title
- word lower string
- word upper string
- length of word
- word numeric
- word alphabet

Let's look at the code for simple feature mapping.

```
# feature mapping for the classifier.

def create_ft(txt):
    return np.array([txt.istitle(), txt.islower(), txt.isupper(),
len(txt),txt.isdigit(),  txt.isalpha()])

#using the above function created to get the mapping of words for train
data.

words = [create_ft(x) for x in data["words"].values.tolist()]

#lets take unique labels
target = data["labels"].values.tolist()

#print few words with array
print(words[:5])

Output:
we got mapping of words as below (for first five words)

[array([0, 1, 0, 5, 0, 1]), array([0, 1, 0, 7, 0, 1]), array([0, 1, 0, 8,
0, 1]), array([0, 1, 0, 1, 0, 1]), array([0, 1, 0, 9, 0, 1])]
```

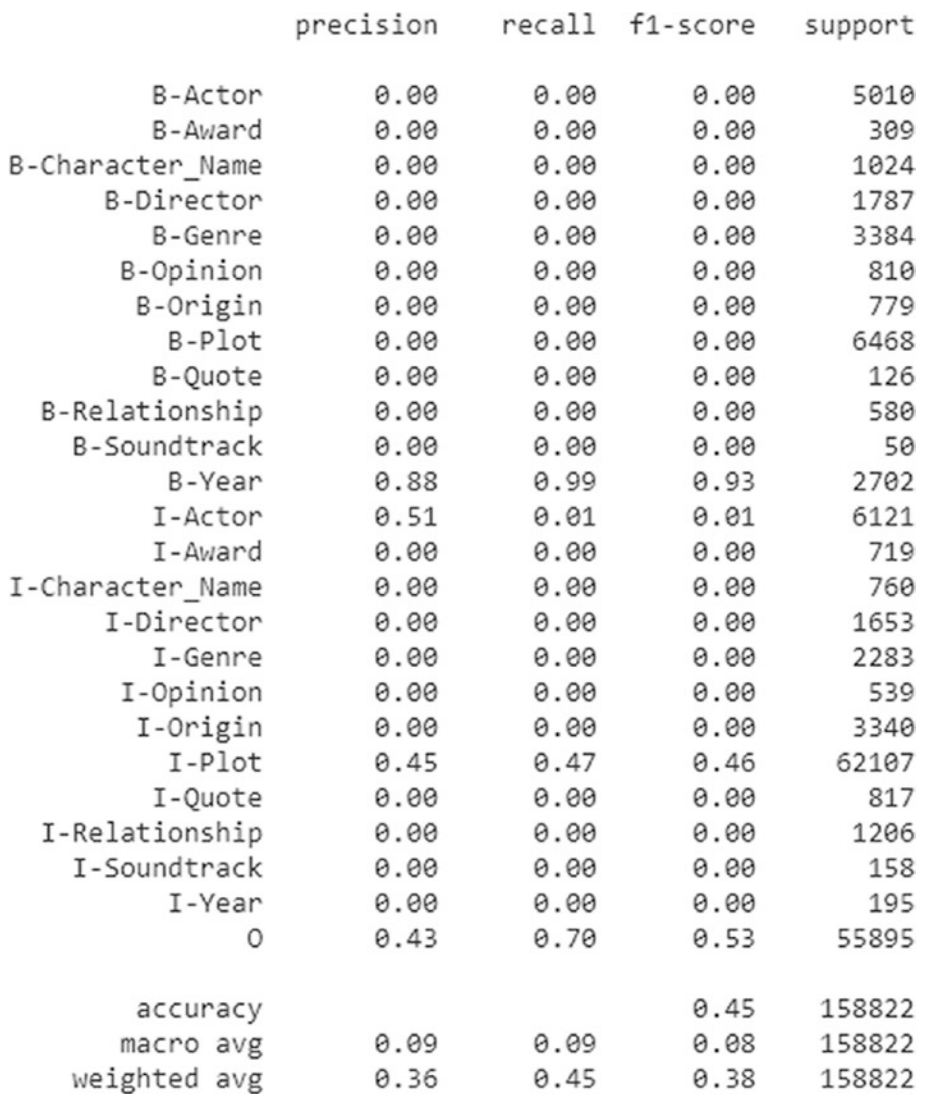
Likewise, we use the function created for the test data.

```
#using the above function created to get the mapping of words for test data.
words1 = [create_ft(x) for x in data1["words"].values.tolist()]
target1 = data1["labels"].values.tolist()
```

Apply five-fold cross validation for the random classifier model and get the results as follows. Next, the cross_val_predict function is used. It is defined in sklearn.

```
#importing package
from sklearn.model_selection import cross_val_predict

# train the RF model
Ner_prediction = cross_val_predict(RandomForestClassifier(n_
estimators=20),X=words, y=target, cv=10)

#import library
from sklearn.metrics import classification_report

#generate report
Accuracy_rpt= classification_report(y_pred= Ner_prediction, y_true=target)
print(Accuracy_rpt)
```

Figure 8-7 shows the classification report.



```
                   precision    recall  f1-score   support

         B-Actor        0.00      0.00      0.00      5010
         B-Award        0.00      0.00      0.00       309
B-Character_Name        0.00      0.00      0.00      1024
      B-Director        0.00      0.00      0.00      1787
         B-Genre        0.00      0.00      0.00      3384
       B-Opinion        0.00      0.00      0.00       810
        B-Origin        0.00      0.00      0.00       779
          B-Plot        0.00      0.00      0.00      6468
         B-Quote        0.00      0.00      0.00       126
  B-Relationship        0.00      0.00      0.00       580
    B-Soundtrack        0.00      0.00      0.00        50
          B-Year        0.88      0.99      0.93      2702
         I-Actor        0.51      0.01      0.01      6121
         I-Award        0.00      0.00      0.00       719
I-Character_Name        0.00      0.00      0.00       760
      I-Director        0.00      0.00      0.00      1653
         I-Genre        0.00      0.00      0.00      2283
       I-Opinion        0.00      0.00      0.00       539
        I-Origin        0.00      0.00      0.00      3340
          I-Plot        0.45      0.47      0.46     62107
         I-Quote        0.00      0.00      0.00       817
  I-Relationship        0.00      0.00      0.00      1206
    I-Soundtrack        0.00      0.00      0.00       158
          I-Year        0.00      0.00      0.00       195
               O        0.43      0.70      0.53     55895

        accuracy                            0.45    158822
       macro avg        0.09      0.09      0.08    158822
    weighted avg        0.36      0.45      0.38    158822
```

Figure 8-7. *Classification report*

Accuracy is 45%, and the overall F1 score is 0.38. The performance is not so good. Now let's further improve the accuracy, leveraging more features, like "pre" and "post" words.

Feature Mapping by Adding More Features

In this case, we consider the features of the next word and previous words corresponding to the given word.

1. Add a tag at the beginning of the sentence if the word starts the sentence.
2. Add a tag at the end of the sentence if the word ends the sentence. Also, consider the characteristics of the previous word and the next word.

Let's import the code for feature mapping.

First, upload the Python file to Colab or any working Python environment, as shown in Figure 8-8.

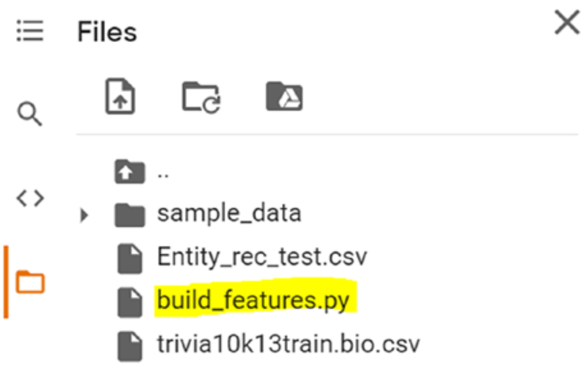


Figure 8-8. *Uploading Python script*

```
from build_features import text_to_features

# function to create features
def text2num(wrd):
    return [text_to_features(wrd, i) for i in range(len(wrd))]

# function to create labels
def text2lbl(wrd):
    return [label for token,label in wrd]
```

Let's prepare the training data features using sent2features function which in turn uses text to features function.

```
X = [text2num(x) for x in sentences]
```

Prepare the training data labels using `text2lbl` because the sentences are in a tuple consisting of words and tags.

```
y = [text2lbl(x) for x in sentences]
```

Similarly, prepare the test data.

```
test_X = [text2num(x) for x in sentences]
```

```
test_y = [text2lbl(x) for x in sentences]
```

Now that the training and test data are ready, let's initialize the model. First, let's initialize and build a CRF model without hyperparameter tuning.

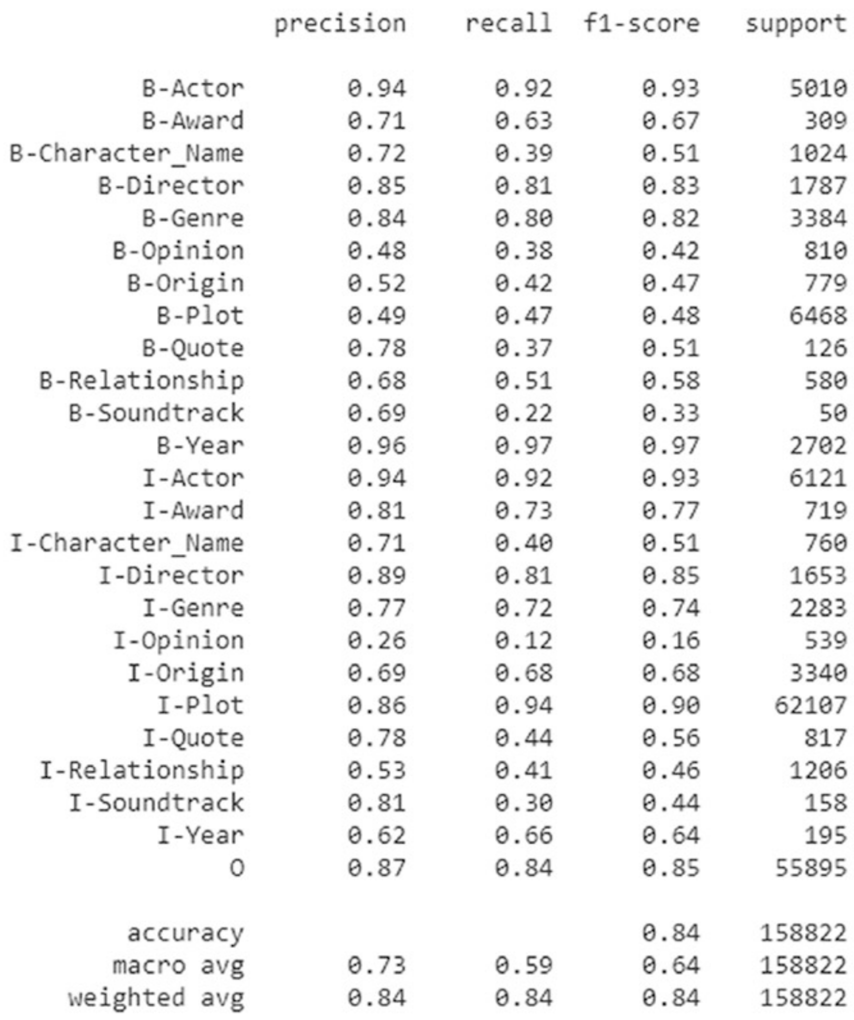
```
#building the CRF model
ner_crf_model = CRF(algorithm='lbfgs',max_iterations=25)
```

```
#tgraining the model with cross validation of 10
ner_predictions = cross_val_predict(estimator= ner_crf_model, X=X, y=y, cv=10)
```

Let's evaluate the model on train data.

```
Accu_rpt = flat_classification_report(y_pred=ner_predictions, y_true=y)
print(Accu_rpt)
```

Figure 8-9 shows the classification report for train data.



```
                  precision    recall  f1-score   support

          B-Actor       0.94      0.92      0.93      5010
          B-Award       0.71      0.63      0.67       309
 B-Character_Name       0.72      0.39      0.51      1024
       B-Director       0.85      0.81      0.83      1787
          B-Genre       0.84      0.80      0.82      3384
        B-Opinion       0.48      0.38      0.42       810
         B-Origin       0.52      0.42      0.47       779
           B-Plot       0.49      0.47      0.48      6468
          B-Quote       0.78      0.37      0.51       126
   B-Relationship       0.68      0.51      0.58       580
     B-Soundtrack       0.69      0.22      0.33        50
           B-Year       0.96      0.97      0.97      2702
          I-Actor       0.94      0.92      0.93      6121
          I-Award       0.81      0.73      0.77       719
 I-Character_Name       0.71      0.40      0.51       760
       I-Director       0.89      0.81      0.85      1653
          I-Genre       0.77      0.72      0.74      2283
        I-Opinion       0.26      0.12      0.16       539
         I-Origin       0.69      0.68      0.68      3340
           I-Plot       0.86      0.94      0.90     62107
          I-Quote       0.78      0.44      0.56       817
   I-Relationship       0.53      0.41      0.46      1206
     I-Soundtrack       0.81      0.30      0.44       158
           I-Year       0.62      0.66      0.64       195
                O       0.87      0.84      0.85     55895

         accuracy                           0.84    158822
        macro avg       0.73      0.59      0.64    158822
     weighted avg       0.84      0.84      0.84    158822
```

Figure 8-9. *Classification report for train data*

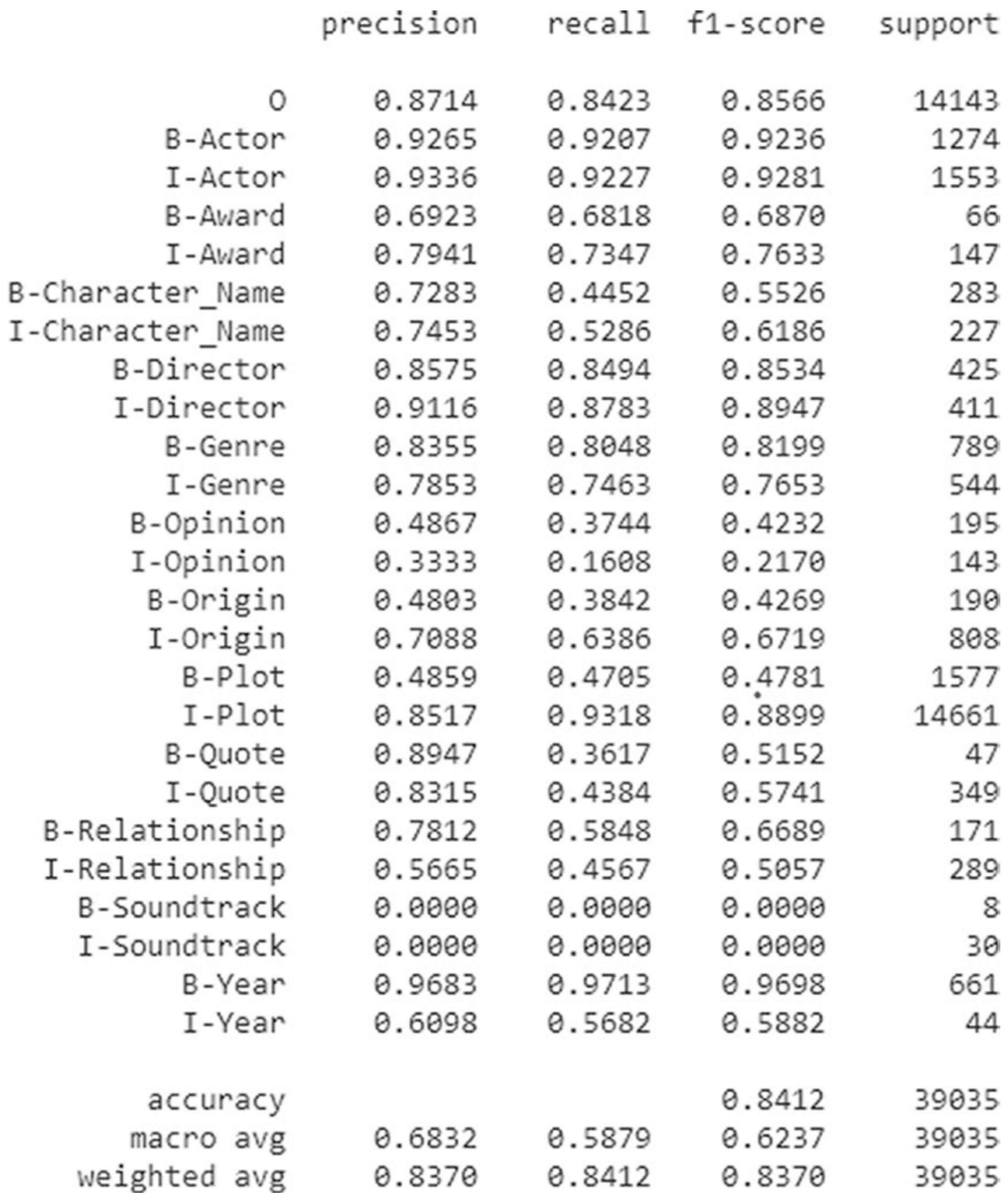
Train data results: the overall F1 score is 0.84, and accuracy is 0.84.

Let's evaluate the model on test data.

```
#building the CRF model
crf_ner = CRF(algorithm='lbfgs',max_iterations=25)

#Fitting model on train data.
crf_ner.fit(X,y)
```

```
# prediction on test data
test_prediction=crf_ner.predict(test_X)
# get labels
lbs=list(crf_ner.classes_)

#get accuracy
metrics.flat_f1_score(test_y,test_prediction,average='weighted',labels=lbs)

Ouptut:
0.8369950502328535

#sort the labels
sorted_lbs=sorted(lbs,key= lambda name:(name[1:],name[0]))

#get classification report
print(metrics.flat_classification_report(test_y,test_prediction,
labels=sorted_lbs,digits=4))
```

Figure 8-10 shows the classification report for the test data.



```
                     precision    recall  f1-score   support

                 O      0.8714    0.8423    0.8566     14143
           B-Actor      0.9265    0.9207    0.9236      1274
           I-Actor      0.9336    0.9227    0.9281      1553
           B-Award      0.6923    0.6818    0.6870        66
           I-Award      0.7941    0.7347    0.7633       147
  B-Character_Name      0.7283    0.4452    0.5526       283
  I-Character_Name      0.7453    0.5286    0.6186       227
        B-Director      0.8575    0.8494    0.8534       425
        I-Director      0.9116    0.8783    0.8947       411
           B-Genre      0.8355    0.8048    0.8199       789
           I-Genre      0.7853    0.7463    0.7653       544
         B-Opinion      0.4867    0.3744    0.4232       195
         I-Opinion      0.3333    0.1608    0.2170       143
          B-Origin      0.4803    0.3842    0.4269       190
          I-Origin      0.7088    0.6386    0.6719       808
            B-Plot      0.4859    0.4705    0.4781      1577
            I-Plot      0.8517    0.9318    0.8899     14661
           B-Quote      0.8947    0.3617    0.5152        47
           I-Quote      0.8315    0.4384    0.5741       349
    B-Relationship      0.7812    0.5848    0.6689       171
    I-Relationship      0.5665    0.4567    0.5057       289
      B-Soundtrack      0.0000    0.0000    0.0000         8
      I-Soundtrack      0.0000    0.0000    0.0000        30
            B-Year      0.9683    0.9713    0.9698       661
            I-Year      0.6098    0.5682    0.5882        44

          accuracy                          0.8412     39035
         macro avg      0.6832    0.5879    0.6237     39035
      weighted avg      0.8370    0.8412    0.8370     39035
```

Figure 8-10. *Classification report for test data*

The test data result shows an overall F1 score of 0.8370 and an accuracy of 0.8412. The accuracy increased by nearly 40% by adding more features, and the F1 score increased by 0.5. We can further increase the accuracy by performing hyperparameter tuning.

Let's take a random sentence and predict the tag using the trained model.

1. Create feature maps, as we did for training data, for input sentences using the word2feature function.
2. Convert into an array and predict using crf.predict(input_vector).

Next, convert each word into features.

```
def text_2_ftr_sample(words):
    return [text_to_features(words, i) for i in range(len(words))]
```

Split the sentences and convert each word into features.

```
#define sample sentence
X_sample=['alien invasion is the movie directed by christoper nollen'.
split()]

#convert to features
X_sample1=[text_2_ftr_sample(x) for x in X_sample]

#predicting the class
crf_ner.predict(X_sample1)
```

The following is the output.

```
[['B-Actor', 'I-Actor', 'O', 'O', 'O', 'O', 'O', 'B-Director',
'I-Director']]
```

Now, let's try the BERT model and check if it performs better than the CRF model.

BERT Transformer

BERT (Bidirectional Encoder Representations from Transformers) is a model that is trained on large data sets. This pretrained model can be fine-tuned as per the requirement and used for different tasks such as sentiment analysis, question answering system, sentence classification, and others. BERT transfers learning in NLP, and it is a state-of-the-art method.

BERT uses transformers, mainly the encoder part. The attention mechanism learns the contextual relationship between words and subwords. Unlike other models, Transformer's encoder learns all sequences at once. The input is a sequence of words (tokens) that are embed into vectors and then proceed to the neural networks. The output is the sequence of tokens that corresponds to the input token of the given sequence.

Let's implement the BERT model.

Import the required libraries.

```
#importing necessary libraries
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, make_scorer

#importing NER models from simple transformers
from simpletransformers.ner import NERModel,NERArgs

#importing libraries for evaluation
from sklearn_crfsuite.metrics import flat_classification_report
from sklearn_crfsuite import CRF, scorers, metrics
```

Let's encode the sentence column using LabelEncoder.

```
#encoding sentence values
data["sentence_id"] = LabelEncoder().fit_transform(data["sentence_id"] )
data1["sentence_id"] = LabelEncoder().fit_transform(data1["sentence_id "] )
```

Let's convert all labels into uppercase.

```
#converting labels to upper string as it is required format
data["labels"] = data["labels"].str.upper()
data1["labels"] = data1["labels"].str.upper()
```

Next, separate the train and test data.

```
X= data[["sentence_id","words"]]
Y =data["labels"]
```

Then, create a train and test data frame.

```
#building up train and test data to dataframe
ner_tr_dt = pd.DataFrame({"sentence_id":data["sentence_id"],"words":data["w
ords"],"labels":data["labels"]})
test_data = pd.DataFrame({"sentence_id":data1["sentence_id"],"words":data1[
"words"],"labels":data1["labels"]})
```

Also, let's store the list of unique labels.

```
#label values
label = ner_tr_dt["labels"].unique().tolist()
```

We need to fine-tune the BERT model so that we can use parameters. Here, we changed epochs number and batch size. To improve the model further, we can change other parameters as well.

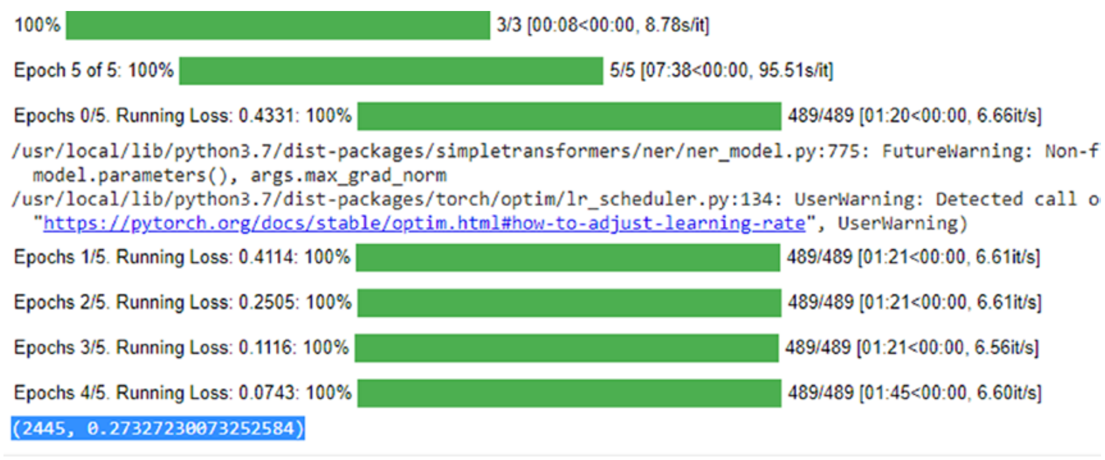
```
#fine tuning our model on custom data
args = NERArgs()

#set the # of epoch
args.num_train_epochs = 2

#learning reate
args.learning_rate = 1e-6
args.overwrite_output_dir =True

#train and evaluation batch size
args.train_batch_size = 6
args.eval_batch_size = 6
```

We now initialize the BERT model.

```
#initializing the model
Ner_bert_mdl= NERModel('bert', 'bert-base-cased',labels=label,args =args)

#training our model
Ner_bert_mdl.train_model(ner_tr_dt,eval_data = test_data,acc=accuracy_
score)
```

The eval_data is the data where loss is calculated. Figure 8-11 shows the output of training the BERT model.

The following is the output.



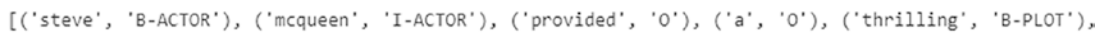
```
100%  3/3 [00:08<00:00, 8.78s/it]
Epoch 5 of 5: 100%  5/5 [07:38<00:00, 95.51s/it]
Epochs 0/5. Running Loss: 0.4331: 100%  489/489 [01:20<00:00, 6.66it/s]
/usr/local/lib/python3.7/dist-packages/simpletransformers/ner/ner_model.py:775: FutureWarning: Non-f
  model.parameters(), args.max_grad_norm
/usr/local/lib/python3.7/dist-packages/torch/optim/lr_scheduler.py:134: UserWarning: Detected call o
  "https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate", UserWarning)
Epochs 1/5. Running Loss: 0.4114: 100%  489/489 [01:21<00:00, 6.61it/s]
Epochs 2/5. Running Loss: 0.2505: 100%  489/489 [01:21<00:00, 6.61it/s]
Epochs 3/5. Running Loss: 0.1116: 100%  489/489 [01:21<00:00, 6.56it/s]
Epochs 4/5. Running Loss: 0.0743: 100%  489/489 [01:45<00:00, 6.60it/s]
(2445, 0.27327230073252584)
```

Figure 8-11. *Training a BERT model*

You can observe a loss of 0.27 in the final epoch.

```
#function to store labels and words of each sentence in list

class sent_generate(object):

    def __init__(self, data):
        self.n_sent = 1.0
        self.data = data
        self.empty = False
        fn_group = lambda s: [(a, b) for a,b in zip(s["words"].values.
        tolist(),s["labels"].values.tolist())]
        self.grouped = self.data.groupby("sentence_id").apply(fn_group)
        self.sentences = [x for x in self.grouped]

#storing words and labels of each sentence in single list of train data

Sent_get= sent_generate(ner_tr_dt)
sentences = Sent_get.sentences

#This is how a sentence will look like.
print(sentences[0])
def txt_2_lbs(sent):
    return [label for token,label in sent]

y_train_group = [txt_2_lbs(x) for x in sentences]
```

Figure 8-12 shows the output for sent_generate function for train data.

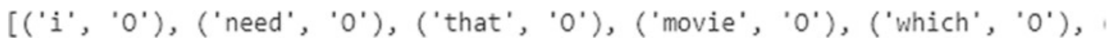


```
[('steve', 'B-ACTOR'), ('mcqueen', 'I-ACTOR'), ('provided', 'O'), ('a', 'O'), ('thrilling', 'B-PLOT'),
```

Figure 8-12. *Output*

```
#storing words and labels of each sentence in single list of test data
Sent_get= sent_generate(test_data)
sentences = Sent_get.sentences

#This is how a sentence will look like.
print(sentences[0])

def txt_2_lbs(sent):
    return [label for token,label in sent]

y_test = [txt_2_lbs(x) for x in sentences]
```

Figure 8-13 shows the output for the sent_generate function for the training data.



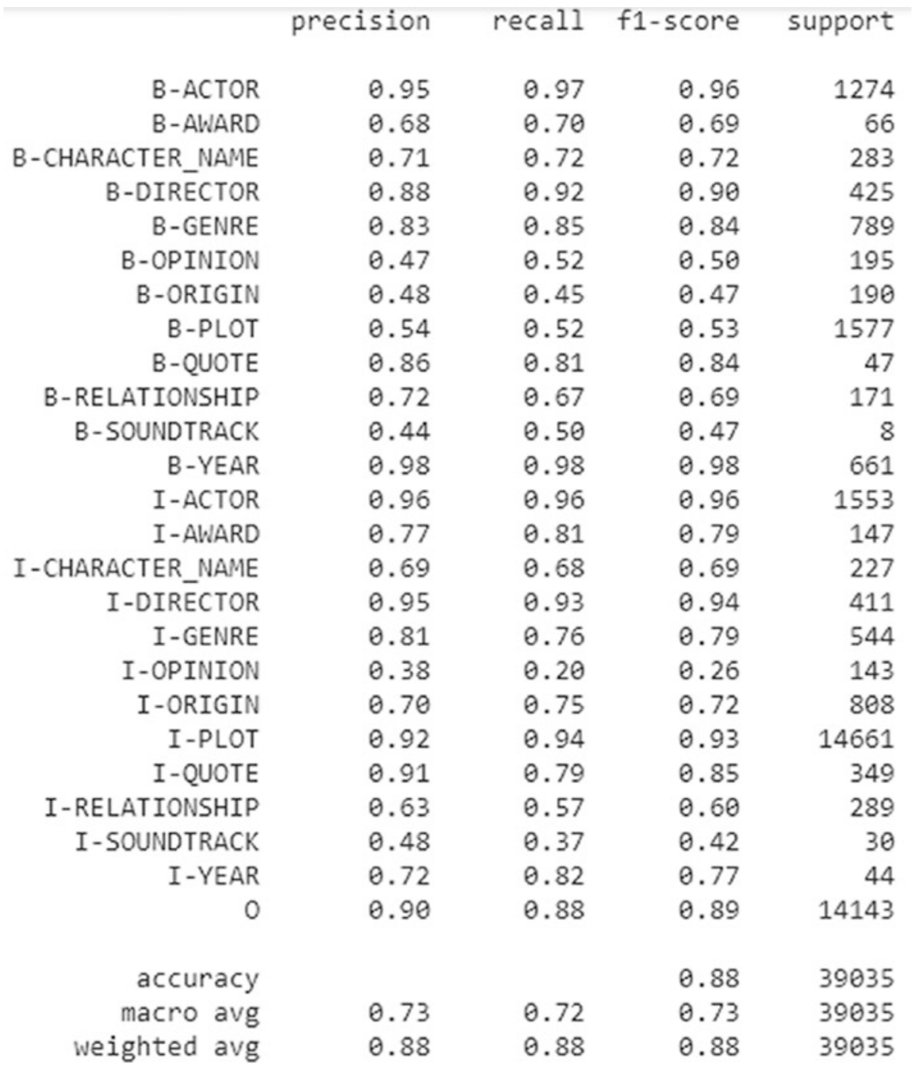
```
[('i', 'O'), ('need', 'O'), ('that', 'O'), ('movie', 'O'), ('which', 'O'),
```

Figure 8-13. *Output*

Let's evaluate the model on test data.

```
#evaluating on test data
result, model_outputs, preds_list = Ner_bert_mdl.eval_model(test_data)

#individual group report
accu_rpt = flat_classification_report(y_pred=preds_list, y_true=y_test)
print(accu_rpt)
```

Figure 8-14 shows the classification report for train data.



```
                   precision    recall  f1-score   support

           B-ACTOR      0.95      0.97      0.96      1274
           B-AWARD      0.68      0.70      0.69        66
  B-CHARACTER_NAME      0.71      0.72      0.72       283
        B-DIRECTOR      0.88      0.92      0.90       425
           B-GENRE      0.83      0.85      0.84       789
         B-OPINION      0.47      0.52      0.50       195
          B-ORIGIN      0.48      0.45      0.47       190
            B-PLOT      0.54      0.52      0.53      1577
           B-QUOTE      0.86      0.81      0.84        47
    B-RELATIONSHIP      0.72      0.67      0.69       171
      B-SOUNDTRACK      0.44      0.50      0.47         8
            B-YEAR      0.98      0.98      0.98       661
           I-ACTOR      0.96      0.96      0.96      1553
           I-AWARD      0.77      0.81      0.79       147
  I-CHARACTER_NAME      0.69      0.68      0.69       227
        I-DIRECTOR      0.95      0.93      0.94       411
           I-GENRE      0.81      0.76      0.79       544
         I-OPINION      0.38      0.20      0.26       143
          I-ORIGIN      0.70      0.75      0.72       808
            I-PLOT      0.92      0.94      0.93     14661
           I-QUOTE      0.91      0.79      0.85       349
    I-RELATIONSHIP      0.63      0.57      0.60       289
      I-SOUNDTRACK      0.48      0.37      0.42        30
            I-YEAR      0.72      0.82      0.77        44
                 O      0.90      0.88      0.89     14143

          accuracy                          0.88     39035
         macro avg      0.73      0.72      0.73     39035
      weighted avg      0.88      0.88      0.88     39035
```

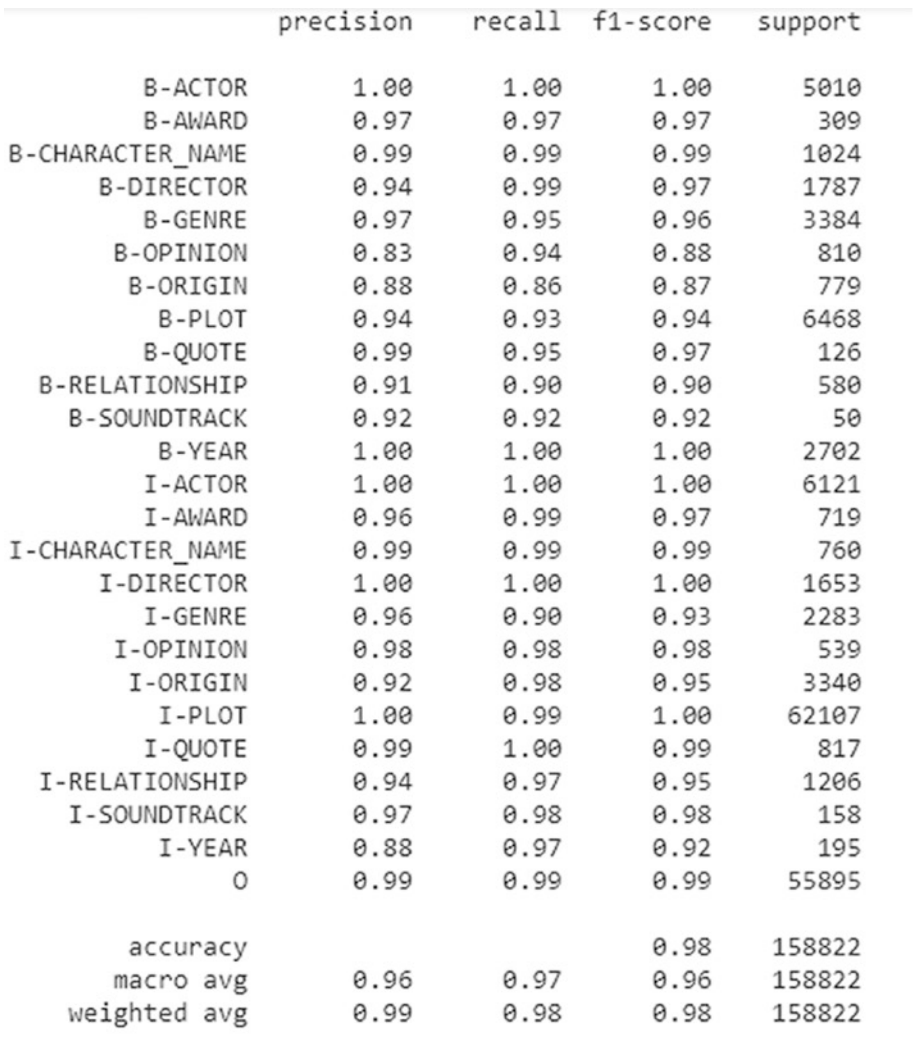
Figure 8-14. *Classification report*

The accuracy of test data is 88%. Let's evaluate the model on train data also to see if there is any overfitting.

```
#evaluating on train data
result_train, model_outputs_train, preds_list_train = Ner_bert_mdl.eval_
model(ner_tr_dt)

#individual group report of train data
```

```
report_train = flat_classification_report(y_pred=preds_list, y_true=y_
train_group)
print(report_train)
```

Figure 8-15 shows the classification report for train data.



```
                  precision    recall  f1-score   support

         B-ACTOR       1.00      1.00      1.00      5010
         B-AWARD       0.97      0.97      0.97       309
B-CHARACTER_NAME       0.99      0.99      0.99      1024
      B-DIRECTOR       0.94      0.99      0.97      1787
         B-GENRE       0.97      0.95      0.96      3384
       B-OPINION       0.83      0.94      0.88       810
        B-ORIGIN       0.88      0.86      0.87       779
          B-PLOT       0.94      0.93      0.94      6468
         B-QUOTE       0.99      0.95      0.97       126
  B-RELATIONSHIP       0.91      0.90      0.90       580
    B-SOUNDTRACK       0.92      0.92      0.92        50
          B-YEAR       1.00      1.00      1.00      2702
         I-ACTOR       1.00      1.00      1.00      6121
         I-AWARD       0.96      0.99      0.97       719
I-CHARACTER_NAME       0.99      0.99      0.99       760
      I-DIRECTOR       1.00      1.00      1.00      1653
         I-GENRE       0.96      0.90      0.93      2283
       I-OPINION       0.98      0.98      0.98       539
        I-ORIGIN       0.92      0.98      0.95      3340
          I-PLOT       1.00      0.99      1.00     62107
         I-QUOTE       0.99      1.00      0.99       817
  I-RELATIONSHIP       0.94      0.97      0.95      1206
    I-SOUNDTRACK       0.97      0.98      0.98       158
          I-YEAR       0.88      0.97      0.92       195
               O       0.99      0.99      0.99     55895

        accuracy                           0.98    158822
       macro avg       0.96      0.97      0.96    158822
    weighted avg       0.99      0.98      0.98    158822
```

Figure 8-15. *Classification report*

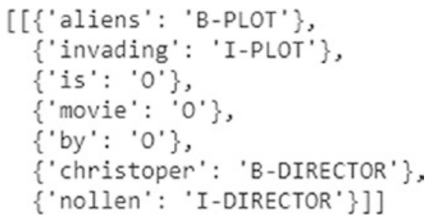
The accuracy of the training data is 98%. Compared to the CRF model, the BERT model is performing great.

Now, let's take a random sentence and predict the tag using the BERT model built.

```
prediction, model_output = Ner_bert_mdl.predict(["aliens invading is movie
by christoper nollen"])
```

Here, the model generates predictions that tag given words from each sentence, and model_output are outputs generated by the model.

Figure 8-16 shows the predictions with the random sentence.



```
[[{'aliens': 'B-PLOT'},
  {'invading': 'I-PLOT'},
  {'is': 'O'},
  {'movie': 'O'},
  {'by': 'O'},
  {'christoper': 'B-DIRECTOR'},
  {'nollen': 'I-DIRECTOR'}]]
```

Figure 8-16. *Model prediction*

Next Steps

- Add more features to the model example combination of words with gives a proper meaning for CRF model.
- The current implementation considers only two hyperparameters in the CV search for the CRF model; however, the CRF model offers more parameters that can be further tuned to improve the performance.
- Use an LSTM neural network and creating an ensemble model with CRF.
- For the BERT model, we can add a CRF layer to get a more efficient network as BERT finds efficient patterns between words.

Summary

We started this project by briefly describing NER and existing libraries to extract topics. Given the limitations, you have to build custom NER models. After defining the problem, we extracted movie names and director names and built an approach to solve the problem.

We tried three models: random forest, CRF, and BERT. We created the features from the text and passed them to RandomForest and CRF models with different features. As you saw, the more features better accuracy.

Instead of building these features manually, we used deep learning algorithms. We took a BERT-based pretrained model and trained NER on top of it. We did not use any manually created features, which is one of the advantages of deep learning models. Another positive point is pretrained models. You know how powerful transfer learning is, and the results of this project also showcase the same thing. We were able to produce good results from NER BERT when compared to handwritten features with CRF.