Named Entity Recognition model on CoNLL 2003 dataset with Web App deployment

In this project we will employ deep learning techniques for developing a **Named Entity Recognition** (NER) model on CoNLL 2003 dataset and further deploy it as a web app. In a nutshell, NER is considered a task of searching and tagging important objects (i.e. named entities) like locations, people or organizations within a chunk of chosen text. From NLP and computational perspective NER is recognized as a sequence labelling problem – given a text input, an algorithm should be capable of assigning aforementioned entity types to particular elements (tokens) of that sequence.

Project will be mainly conducted in **Python**, **Tensorflow** with **Keras**, with an inclusion of **Tensorflow.js** and **JavaScript** (required for web model deployment) and simple **html** script. Theoretical steps might be summarized by the following workflow:

1. Explore data, extract inputs and corresponding tags (labels) from provided CoNLL 2003 txt files.
2. Conduct any required text preprocessing, if necessary.
3. Tokenize and create vocabulary.
4. Text vectorization (representing texts in numerical form) of inputs and tags.
5. Design LSTM based model architecture with a CNN extension.
6. Train model with hyperparameter tuning.
7. Evaluate performance on test data to compare against benchmarks.
8. Design simple html website powered by Tensorflow.js converted Keras model.
9. Website tests and final deployment.

Import relevant libraries, functions, classes and initial config file

In [1]:

```python
import json
import os

import joblib
import tensorflow as tf
from evaluation.conlleval import evaluate
from model_tf.model_tuner import tf_set_memory_growth

from utils.encoders import SequentialLabelEncoder
from utils.inference import getPredictedNER
from utils.text import (CustomTokenizer, TextPreprocessor, read_txt_file,
                        write_js_from_dict)

with open('config.json') as config_file:
    conf = json.load(config_file)
```

Config file contains paths to relevant datasets and two variables defining maximum sequence length and maximum word length for model input. The last two variables are set to rather standard lengths so that most sentences and words do fit within specified ranges (otherwise they will be truncated). Config will be further extended by vocabulary sizes and number of entity tag classes (as those are required for building model).

In [2]:

```python
print(json.dumps(conf, indent=2))
```

```
{
  "PATH_TRAIN": "data/conll2003/train.txt",
  "PATH_VALID": "data/conll2003/valid.txt",
  "PATH_TEST": "data/conll2003/test.txt",
  "MAX_SEQ_LEN": 64,
  "MAX_WRD_LEN": 16
}
```

CoNLL 2003 dataset

An English version of CoNLL 2003 dataset will be used. The dataset consists of Reuters news stories gathered between 1996 and 1997 and is widely exploited across many NER researchers to produce state of the art results. The dataset is divided into train, validation and test files with a total of over 22k annotated sentences. Each file contains four single space separated columns with the first column being a word (token) of a given sentence (sentences are encoded in a row-wise manner) and the last one representing an annotated entity, i.e. label to predict for a particular word. Second and third ones are part-of-speech and syntactic tags respectively and are of our less interest in the NER context, at least for this particular project.

Entities are split into four groups: person (B-PER, I-PER), location (B-LOC, I-LOC), organization (B-ORG, I- ORG) and miscellaneous (B-MISC, I-MISC). There is also a separate label O for no entity types. The I- prefix is applied in case of multi word entities of the same type to represent words following the first one. For example [European, Commission] would be tagged as [B-ORG, I-ORG]. More details on this dataset can be found here: https://www.clips.uantwerpen.be/conll2003/ner/.

We can print out a few sentences from **train.txt** file for a more thorough inspection. Valid and test files are organized identically.

In [3]:

```python
with open(conf['PATH_TRAIN']) as f:
    for i, line in enumerate(f):
        if i < 48:
            print(line.rstrip())
```

```
-DOCSTART- -X- -X- O

EU NNP B-NP B-ORG
rejects VBZ B-VP O
German JJ B-NP B-MISC
call NN I-NP O
to TO B-VP O
boycott VB I-VP O
British JJ B-NP B-MISC
lamb NN I-NP O
. . O O

Peter NNP B-NP B-PER
Blackburn NNP I-NP I-PER

BRUSSELS NNP B-NP B-LOC
1996-08-22 CD I-NP O

The DT B-NP O
European NNP I-NP B-ORG
Commission NNP I-NP I-ORG
said VBD B-VP O
on IN B-PP O
Thursday NNP B-NP O
it PRP B-NP O
```

```
disagreed VBD B-VP O
with IN B-PP O
German JJ B-NP B-MISC
advice NN I-NP O
to TO B-PP O
consumers NNS B-NP O
to TO B-VP O
shun VB I-VP O
British JJ B-NP B-MISC
lamb NN I-NP O
until IN B-SBAR O
scientists NNS B-NP O
determine VBP B-VP O
whether IN B-SBAR O
mad JJ B-NP O
cow NN I-NP O
disease NN I-NP O
can MD B-VP O
be VB I-VP O
transmitted VBN I-VP O
to TO B-PP O
sheep NN B-NP O
. . O O
```

Load data and preprocess

To load data for further processing we will use a custom function `read_txt_file` available in `utils.text` module. This function simply returns two nested lists with the first one gathering sentences tokens and the latter representing corresponding labels.

In [4]:

```python
sentences_train, labels_train = read_txt_file(conf['PATH_TRAIN'])
sentences_valid, labels_valid = read_txt_file(conf['PATH_VALID'])
sentences_test, labels_test = read_txt_file(conf['PATH_TEST'])
print('Tokens and labels for the first sentence from training file: \n {}, \n {} \n'
    sentences_train[0], labels_train[0]))
print('Number of train, valid and test instances are {}, {} and {} respectively.\n'.
    len(sentences_train), len(sentences_valid), len(sentences_test)))
```

```
Tokens and labels for the first sentence from training file:
 ['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.'],
 ['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O', 'O']

Number of train, valid and test instances are 14041, 3250 and 3453 respectively.
```

As previously mentioned, sentences are already tokenized but to preprocess data and fit a tokenizer to create a generalizable algorithm we will need to proceed on standard, joined texts (this also mimics a production like behaviour as an end user will type in a text, not separate tokens). Thus, `texts` variable will be created on training data to further obtain vocabulary.

In [5]:

```python
texts = [' '.join(s) for s in sentences_train]
print(texts[0])
```

```
EU rejects German call to boycott British lamb .
```

Before tokenization we will preprocess texts with `TextPreprocessor` class from `utils.text`. This class separates apostrophes and punctuation from words as we would like to ensure that *John's* will not be treated as a different token than *John*. This is also in line with CoNLL 2003 dataset convention. For this project we will not remove any punctuation to preserve the full sequential sentence structure for the model input.

In [6]:
```
text_prc = TextPreprocessor(separate_apostrophes=True, separate_punctuation=True)
texts = text_prc(texts)
sample_text = "Jack's son lives in London. I don't like him."
print("Before: '{}' \nand after: '{}' \napplying TextPreprocessor.\n".format(sample_
```

```
Before: 'Jack's son lives in London. I don't like him.'
and after: 'Jack 's son lives in London . I do n't like him .'
applying TextPreprocessor.
```

Tokenization

In this step we will define a `CustomTokenizer` class (based on `tf.keras.preprocessing.text.Tokenizer`) available in `utils.text`. This class provides a few important utilities for creating a proper numerical model input:

1. Tokenize input on word level and apply lower case letters.
2. Tokenize each word token on character level but without lower case letters.
3. Add [UNK] and [PAD] (unknown and padding) tokens to vocabulary.
4. Return word and character level token ids with additional mask indicating padded sequences.

Furthermore, `CustomTokenizer` class might be applied to pure text input (e.g. during inference), list of texts or list of lists of texts (the latter is a CoNLL 2003 case).

In [7]:
```
tokenizer = CustomTokenizer(char_level=True, lower=True)
tokenizer.fit(texts, max_seq_len=conf['MAX_SEQ_LEN'], max_word_len=conf['MAX_WRD_LEN
X_train = tokenizer.transform(sentences_train)
X_valid = tokenizer.transform(sentences_valid)
X_test = tokenizer.transform(sentences_test)
```

We can then inspect for example `X_train` object and its first instance to verify the result from applying tokenizer's `transform` method.

In [8]:
```
print('Word tokens: \n{}\n'.format(sentences_train[0]))
print('Token ids on word level input: \n{}\n'.format(X_train[0][0]))
print('Token ids on character level input (for first 10 word tokens): \n{}\n'.format
print('Input mask: \n{}\n'.format(X_train[2][0]))
```

```
Word tokens:
['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.']

Token ids on word level input:
[   959 10390    216    645      8  3966    226  5756      2     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0     0     0     0     0     0     0     0     0
      0     0     0     0]

Token ids on character level input (for first 10 word tokens):
[[38 60  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 9  3 64  3 14  5 10  0  0  0  0  0  0  0  0  0]
 [54  3  9 16  4  6  0  0  0  0  0  0  0  0  0  0]
 [14  4 11 11  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [23  8 21 14  8  5  5  0  0  0  0  0  0  0  0  0]
 [46  9  7  5  7 10 13  0  0  0  0  0  0  0  0  0]
 [11  4 16 23  0  0  0  0  0  0  0  0  0  0  0  0]
 [20  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]

Input mask:
[1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

A few important things arise from matrices provided above. First off, both word and mask inputs are **2D** matrices while character level input is represented by a **3D** matrix - each sentence is a sequence of **MAX_SEQ_LEN** words (second dimension) and each word is a sequence of **MAX_WRD_LEN** characters (third dimension). To verify that we will provide all shapes in the cell below. Secondly, input mask has **1** in the position of any valid token (i.e. token that will be considered during model training or evaluation) and **0** otherwise (for padded tokens).

In [9]:
```
print('Shapes for word level input, character level input and input mask are {}, {}
    X_train[0].shape, X_train[1].shape, X_train[2].shape))
```

```
Shapes for word level input, character level input and input mask are (14041, 64),
(14041, 64, 16) and (14041, 64) respectively.
```

Encode labels (tags)

The second step of vectorizing text is label encoding, i.e. assigning ids to corresponding labels (e.g. assign **0** id to **B-LOC** tag). For this purpose we will utilise `SequentialLabelEncoder` class (from `utils.encoders`) which uses a `sklearn.preprocessing.LabelEncoder` underneath but has two extra features: it pads sequences to a given length and additionally returns a mask (similar to the one from tokenization section). Thus, we end up with an **y** array that has consistent shape across all inputs and serves as a model output.

In [10]:
```
lb_enc = SequentialLabelEncoder()
lb_enc.fit(labels_train, max_seq_len=conf['MAX_SEQ_LEN'])
labels_dict = dict(enumerate(lb_enc.le.classes_))
print(json.dumps(labels_dict, indent=2))
```

```
{
  "0": "B-LOC",
  "1": "B-MISC",
  "2": "B-ORG",
  "3": "B-PER",
  "4": "I-LOC",
  "5": "I-MISC",
  "6": "I-ORG",
  "7": "I-PER",
  "8": "O"
}
```

In [11]:
```
y_train, y_train_mask = lb_enc.transform(labels_train)
y_valid, y_valid_mask = lb_enc.transform(labels_valid)
y_test, y_test_mask = lb_enc.transform(labels_test)
print('Ids of labels with corresponding mask:\n{},\n{},\n{}'.format(labels_train[0],
```

```
Ids of labels with corresponding mask:
['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O', 'O'],
[2 8 1 8 8 8 1 8 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0],
[1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Note that padding value (**0**) is the same as one of labels ids but it is not an issue as we are using mask and **0** will be only considered in cases where it represents an entity.

Save objects

After applying all steps described so far we are ready to save (or update) different objects that will be further used for model trainig, inference and web app deployment:

1. Data for model development and evaluation.
2. Text preprocessor, tokenizer and label encoder (for inference).
3. Vocabulary and label dictionaries (those are saved with `write_js_from_dict` function as **js** files for web app deployment).
4. Config file updated with vocabularies and labels lengths.

In [12]:
```
data = {'train': (X_train, y_train, y_train_mask),
        'valid': (X_valid, y_valid, y_valid_mask),
        'test': (X_test, y_test, y_test_mask)}
joblib.dump(data, 'data/data.joblib')
```

Out[12]:
```
['data/data.joblib']
```

In [13]:
```
joblib.dump(text_prc, 'inference/text_preprocessor.joblib')
joblib.dump(tokenizer, 'inference/tokenizer.joblib')
joblib.dump(lb_enc.le, 'inference/label_encoder.joblib')
```

Out[13]:
```
['inference/label_encoder.joblib']
```

In [14]:
```
write_js_from_dict('web-app/vocabs/wordVocab.js',
                   tokenizer.word_tokenizer.word_index,
                   const_name='wordVocab')
write_js_from_dict('web-app/vocabs/charVocab.js',
                   tokenizer.char_tokenizer.word_index,
                   const_name='charVocab')
write_js_from_dict('web-app/vocabs/labels.js',
                   labels_dict,
                   const_name='labels')
```

In [15]:
```
conf['WORD_VOCAB_SIZE'] = len(tokenizer.word_tokenizer.word_index)
conf['CHAR_VOCAB_SIZE'] = len(tokenizer.char_tokenizer.word_index)
conf['NUM_CLASSES'] = len(lb_enc.le.classes_)
print(json.dumps(conf, indent=2))

with open('config.json', 'w') as config_file:
    json.dump(conf, config_file, indent=2)
```

```
{
  "PATH_TRAIN": "data/conll2003/train.txt",
  "PATH_VALID": "data/conll2003/valid.txt",
  "PATH_TEST": "data/conll2003/test.txt",
  "MAX_SEQ_LEN": 64,
  "MAX_WRD_LEN": 16,
  "WORD_VOCAB_SIZE": 17723,
  "CHAR_VOCAB_SIZE": 87,
  "NUM_CLASSES": 9
}
```

Develop model

Main deep learning model will be developed in Tensorflow and will be comprised of a few components:

1. **Embedding** layers with pretrained **GloVe** embeddings for words (100 dimensional, available under https://nlp.stanford.edu/projects/glove/) and trained from scratch for characters.
2. **1D Convolutional** layer with global max pooling to extract character features on word token level.
3. **Bidirectional LSTM** layer that will sequentially proccess both word and character features (concatenated).
4. **Dense** layer (directly connected to LSTM output) with a **relu** activation and **dropout** applied.
5. **Dense** layer with a **softmax** activation to output entity tags probabilities for each word token.

As we are dealing with a **sequence to sequence** problem, we need a separate output for each sequence element (i.e. word token). That is why **Time Distributed** layer will be utilised. The model architecture also incorporates **mask** level input to ignore padded sequences during backpropagation. **Sample weights** are introduced in model fit procedure (and evaluation as well) to mask artificial (padded) labels (therefore weighted loss will not account for 0 weighted instances).

For hyperparameter tuning with `keras-tuner` an efficient **Bayesian Optimization** algorithm will be used with a validation **weighted categorical crossentropy** being monitored as a primary metric (a weighted version of loss is of our main concern as it doesn't take into account padded labels). We run **100 trials** of hp tuning with a maximum of **50 epochs** per trial (**early stopping** is introduced as well) and a fixed **batch size** of **64**. **Hyperparameters** to be tuned are: character embedding dimension, number of dense and lstm units, number of convolutional filters, convolutional kernel size, dropout and learning rate.

Final model is saved in h5 and tfjs format (for web app deployment). The whole thing is run with a **GPU** support. Tuning script is available as `model_tuner.py` and might be inspected below.

In [16]:
```
!pygmentize model_tf/model_tuner.py
```

```
# -*- coding: utf-8 -*-

"""Hyperparameter tuning."""


import json

import joblib
import keras_tuner as kt
import tensorflow as tf
import tensorflowjs as tfjs
from tensorflow.keras.layers import (Bidirectional, Concatenate, Conv1D,
                                     Dense, Dropout, Embedding, GlobalMaxPool1D,
                                     Input, LSTM, TimeDistributed)

from utils.text import get_glove_embedding_matrix


def tf_set_memory_growth():
    """Set memory growth option for GPU device."""
    gpu_devices = tf.config.experimental.list_physical_devices('GPU')
    if len(gpu_devices) > 0:
        tf.config.experimental.set_memory_growth(gpu_devices[0], True)


def build_model(hp):
    """Build Keras model with hyperparameters."""

    # hyperparameters
    dns_units = hp.Int('dns_units', min_value=160, max_value=256, step=4)
    lstm_units = hp.Int('lstm_units', min_value=64, max_value=160, step=4)
    ch_emb_dim = hp.Int('ch_emb_dim', min_value=8, max_value=32, step=2)
    conv_filters = hp.Int('conv_filters', min_value=16, max_value=48, step=2)
    conv_kernel = hp.Choice('conv_kernel', values=[2, 3, 4, 5])
    drp = hp.Float('drp', min_value=0.2, max_value=0.5, step=0.025)
    lr = hp.Choice('lr', values=[5e-4, 7.5e-4, 1e-3])

    # mask input on word sequence level
    mask_inp = Input(shape=(conf['MAX_SEQ_LEN'],), name="mask_input")

    # word level input
    word_inp = Input(shape=(conf['MAX_SEQ_LEN'],), name='word_input')
    word_emb = Embedding(input_dim=conf['WORD_VOCAB_SIZE'],
                         output_dim=EMBEDDING_MATRIX.shape[1], mask_zero=True,
                         input_length=conf['MAX_SEQ_LEN'],
                         weights=[EMBEDDING_MATRIX], trainable=False,
                         name='word_embedding')(word_inp)

    # character level input
    char_inp = Input(shape=(conf['MAX_SEQ_LEN'], conf['MAX_WRD_LEN']),
                     name="character_input")
    char_emb = TimeDistributed(
        Embedding(input_dim=conf['CHAR_VOCAB_SIZE'], output_dim=ch_emb_dim,
                  input_length=conf['MAX_WRD_LEN']),
        name="character_embedding")(char_inp)
    conv1d = TimeDistributed(
        Conv1D(filters=conv_filters, kernel_size=conv_kernel,
               strides=1, padding='same', activation='tanh'),
        name="1d_character_convolution")(char_emb)
    char_features = TimeDistributed(
        GlobalMaxPool1D(),
        name="global_max_pooling")(conv1d)

    # concatenation and output
    conc = Concatenate(name='concatenation')([word_emb, char_features])
    lstm = Bidirectional(LSTM(lstm_units, return_sequences=True),
                         name='bi_lstm')(conc)
    dns = TimeDistributed(
        Dense(dns_units, activation='relu'),
        name='dense')(lstm)
    dns_drop = Dropout(drp, name='dense_dropout')(dns)
    out = TimeDistributed(
        Dense(conf['NUM_CLASSES'], activation='softmax'),
        name='output')(dns_drop, mask=mask_inp)

    model = tf.keras.Model([word_inp, char_inp, mask_inp], out)
    model.compile(optimizer=tf.keras.optimizers.Adam(lr),
                  loss='sparse_categorical_crossentropy',
                  weighted_metrics=['sparse_categorical_crossentropy',
                                    'sparse_categorical_accuracy'])

    return model


if __name__ == '__main__':
    # load config file, development data and pretrained embeddings
    # for building model and hyperparameter tuning with Bayesian Optimization;
    # save best model in h5 format and convert to tfjs for web app deployment;
    # evaluate on train, validation and test datasets;

    tf_set_memory_growth()

    with open('config.json') as config_file:
        conf = json.load(config_file)

    data = joblib.load('data/data.joblib')
    X_train, y_train, y_train_mask = data['train']
    X_valid, y_valid, y_valid_mask = data['valid']
    X_test, y_test, y_test_mask = data['test']

    word_index = (joblib.load('inference/tokenizer.joblib')
                  .word_tokenizer.word_index)
    EMBEDDING_MATRIX = get_glove_embedding_matrix(
        'data/glove.6B.100d.txt', word_index
    )

    es_callback = tf.keras.callbacks.EarlyStopping(
        monitor='val_sparse_categorical_crossentropy', min_delta=0,
        patience=2, verbose=0, mode='min', baseline=None,
        restore_best_weights=True
        )

    tuner = kt.BayesianOptimization(
        hypermodel=build_model, objective='val_sparse_categorical_crossentropy',
        max_trials=100, num_initial_points=10, directory='model_tf',
        project_name='hptuning'
        )

    fit_args = {'x': X_train, 'y': y_train, 'sample_weight': y_train_mask,
                'validation_data': (X_valid, y_valid, y_valid_mask),
                'epochs': 50, 'batch_size': 64, 'shuffle': True,
                'verbose': 2, 'callbacks': [es_callback]}

    tuner.search(**fit_args)
    best_hparams = tuner.get_best_hyperparameters(num_trials=1)[0]
    model = tuner.hypermodel.build(best_hparams)
    print(model.summary())
    model.fit(**fit_args)

    model.save('inference/h5_model/model.h5')
    tfjs.converters.save_keras_model(model, 'web-app/tfjs_model')

    print('\n')
    print('Categorical crossentropy and Accuracy for training data: {}.'.format(
        model.evaluate(x=X_train, y=y_train, sample_weight=y_train_mask, verbose=0)[
1:]))
    print('Categorical crossentropy and Accuracy for validation data: {}.'.format(
        model.evaluate(x=X_valid, y=y_valid, sample_weight=y_valid_mask, verbose=0)[
1:]))
    print('Categorical crossentropy and Accuracy for test data: {}.'.format(
        model.evaluate(x=X_test, y=y_test, sample_weight=y_test_mask, verbose=0)[1
:]))
```

We will run the background script from bash and save the log for further examination. After the process is finished, train, validation and test datasets will be evaluated and metrics monitored during training will be printed out from log file.

In [17]:
```
os.system('nohup python -m model_tf.model_tuner > model_tf/model_tuner.log &')
```

Out[17]:
```
0
```

In [18]:
```
with open('model_tf/model_tuner.log') as log_file:
    for line in log_file:
        if 'Categorical crossentropy and Accuracy for' in line:
            print(line.rstrip())
```

```
Categorical crossentropy and Accuracy for training data: [0.04258184880018234, 0.987
6053333282471].
Categorical crossentropy and Accuracy for validation data: [0.07042563706636429, 0.9
788177609443665].
Categorical crossentropy and Accuracy for test data: [0.10563959181308746, 0.9671876
430511475].
```

Evaluate with F1 score

Developed model will be additionally evaluated with **F1 score**, a typical metric for assessing NER systems, which is a weighted average of both recall and precision. Precision is the percentage of named entities found by the learning system that are correct while recall is the percentage of named entities present in the corpus that are found by the system. An evaluation standard is that we operate on entity level, therefore *[New, York] [B-LOC, I-LOC]* becomes one instance (*LOC*). An exact match only is considered a correct prediction therefore any partial matches where algorithm correctly identifies an entity existence but predicts the wrong type are discarded. Recall and precision metrics are computed globally by counting the total number of true positives, false negatives and false positives, therefore F1 metric is **micro-averaged**. https://www.clips.uantwerpen.be/conll2000/chunking/output.html provides an official evaluation Perl script (`conlleval`) that will be used in our case and can be found in `evaluation` folder. For that purpose we need to prepare a proper input in a txt file with token, true label and predicted label in each line and sentences separated by empty lines. `create_eval_file` from `utils.text` does exactly that. This function is preceeded by `indices_to_labels` (from `utils.encoders`) to obtain token corresponding labels from model predicted ids. To automate those activities a wrapper Python function called `evaluate` (from `evaluation.conlleval`) will be used.

In [19]:
```
sentences_test, labels_test = read_txt_file(conf['PATH_TEST'])
label_encoder = joblib.load('inference/label_encoder.joblib')
X_test, _, y_test_mask = joblib.load('data/data.joblib')['test']

tf_set_memory_growth()
model = tf.keras.models.load_model('inference/h5_model/model.h5')

evaluate(sentences_test, labels_test, label_encoder, X_test, y_test_mask, model)
```

```
processed 46324 tokens with 5626 phrases; found: 5923 phrases; correct: 4682.
accuracy:  96.72%; precision:  79.05%; recall:  83.22%; FB1:  81.08
              LOC: precision:  80.92%; recall:  90.37%; FB1:  85.38  1855
             MISC: precision:  58.03%; recall:  78.21%; FB1:  66.63  946
              ORG: precision:  80.24%; recall:  71.64%; FB1:  75.70  1483
              PER: precision:  87.98%; recall:  90.01%; FB1:  88.98  1639
```

Obtained F1 Score (**FB1: 81.08**) might be considered satisfactory as it is way above the baseline reported for CoNLL 2003 dataset (**59.61**, https://aclweb.org/aclwiki/CONLL-

2003_(State_of_the_art)) and is only a few percentage points lower than other similar deep learning approaches available under https://paperswithcode.com/sota/named-entity-recognition-ner-on-conll-2003. An important thing to note is that the main idea of this project is not to push the state of the art results but rather to create a decently performing, moderately simple and generalizable algorithm that would be further deployed in a browser. To boost the score one may use a more complex architecture (for example transformer based), higher dimensional embeddings, lexicons, larger external vocabulary from pretrained models (e.g. BERT like) or pretrained models themselves. External vocabulary would be also very beneficial in terms of preventing overfitting as using the vocabulary only from the training data might naturally skew the model.

Deploy Web App

All components required for developing and deploying web application powered by Tensorflow model are available in `web-app` folder. Those include in particular:

1. `ner-web-app.html` and `style.css` files with the whole web design.
2. `tfjs_model` folder with Tensorflow.js converted model object.
3. `vocabs` folder with word/character vocabularies and dictionary with labels (all stored in js files).
4. `javascript` folder with scripts for loading model, handling text input, getting predictions and specific text formatting for output. Functions for (pre)processing text (separating apostrophes, tokenization etc.) are similar in behaviour to those used in Python during model development.

To ensure that deployed model works properly we will compare the browser output with a Python model in *pseudo* inference mode (*pseudo* as it only serves as an example and is not deployed actually). Therefore a simple `getPredictedNER` class that takes text as input and returns entity tags is introduced.

In [20]:
```
!pygmentize utils/inference.py
```

```
# -*- coding: utf-8 -*-

"""Running model in inference mode."""


from dataclasses import dataclass

import numpy as np
from tensorflow.keras import Model as KerasModel

from utils.text import CustomTokenizer, TextPreprocessor


@dataclass
class getPredictedNER:
    """End to end NER prediction for given text."""
    preprocessor: TextPreprocessor
    tokenizer: CustomTokenizer
    model: KerasModel
    labels: list

    def __call__(self, input_text):
        input_text = self.preprocessor(input_text)
        input_X = self.tokenizer.transform(input_text)
        input_mask = np.squeeze(input_X[-1]).astype(bool)
        input_tokens = self.tokenizer.get_original_tokens(input_text)
        pred_ids = np.argmax(self.model.predict(input_X), axis=-1)
        pred_ids = np.squeeze(pred_ids)[input_mask]
        pred_cls = [self.labels[idx] for idx in pred_ids]
        ner_output = [(tkn, ner) for tkn, ner
                      in zip(input_tokens, pred_cls)]

        return ner_output
```

For that class, text preprocessor, tokenizer, Keras model and labels should be provided - all available in `inference` folder. We will also get two sample texts from different sources to be used in this example.

In [21]:

```
# all required objects for inference are already available in the worksapce
# though we still load them to check for completeness
text_prc = joblib.load('inference/text_preprocessor.joblib')
tokenizer = joblib.load('inference/tokenizer.joblib')
tf_set_memory_growth()
model = tf.keras.models.load_model('inference/h5_model/model.h5')
label_encoder = joblib.load('inference/label_encoder.joblib')

get_ner = getPredictedNER(text_prc, tokenizer, model, label_encoder.classes_.tolist(
```

In [22]:

```
txt1 = "Abraham Lincoln (February 12, 1809 – April 15, 1865) was an American lawyer
get_ner(txt1)
```

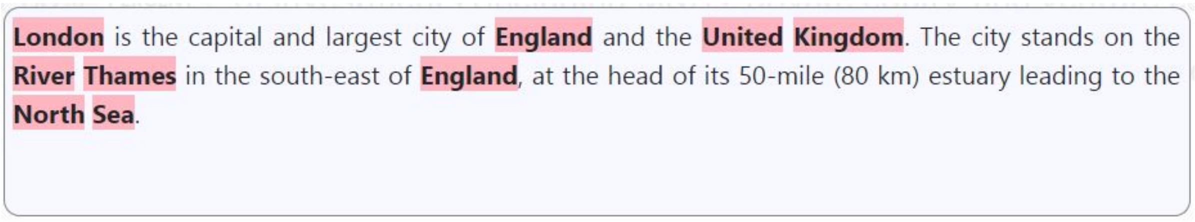
Out[22]:

```
[('Abraham', 'B-PER'),
 ('Lincoln', 'I-PER'),
 ('(', 'O'),
 ('February', 'O'),
 ('12', 'O'),
 (',', 'O'),
 ('1809', 'O'),
 ('-', 'O'),
 ('April', 'O'),
 ('15', 'O'),
 (',', 'O'),
 ('1865', 'O'),
 (')', 'O'),
 ('was', 'O'),
 ('an', 'O'),
 ('American', 'B-MISC'),
 ('lawyer', 'O'),
 ('and', 'O'),
 ('statesman', 'O'),
 ('who', 'O'),
 ('served', 'O'),
 ('as', 'O'),
 ('the', 'O'),
 ('16th', 'O'),
 ('president', 'O'),
 ('of', 'O'),
 ('the', 'O'),
 ('United', 'B-LOC'),
 ('States', 'I-LOC'),
 ('from', 'O'),
 ('1861', 'O'),
 ('until', 'O'),
 ('his', 'O'),
 ('assassination', 'O'),
 ('in', 'O'),
 ('1865', 'O'),
 ('.', 'O'),
 ('Lincoln', 'B-LOC'),
 ('led', 'O'),
 ('the', 'O'),
 ('nation', 'O'),
 ('through', 'O'),
 ('the', 'O'),
 ('American', 'B-MISC'),
 ('Civil', 'I-MISC'),
 ('War', 'I-MISC'),
 (',', 'O'),
 ('the', 'O'),
 ('country', 'O'),
 ("'s", 'O'),
 ('greatest', 'O'),
 ('moral', 'O'),
 (',', 'O'),
 ('cultural', 'O'),
 (',', 'O'),
 ('constitutional', 'O'),
 (',', 'O'),
 ('and', 'O'),
 ('political', 'O'),
 ('crisis', 'O'),
 ('.', 'O')]
```

Abraham **Lincoln** (February 12, 1809 – April 15, 1865) was an **American** lawyer and statesman who served as the 16th president of the **United** **States** from 1861 until his assassination in 1865. **Lincoln** led the nation through the **American** **Civil** **War**, the country's greatest moral, cultural, constitutional, and political crisis.

In [23]:
```
txt2 = "London is the capital and largest city of England and the United Kingdom. Th
get_ner(txt2)
```

Out[23]:
```
[('London', 'B-LOC'),
 ('is', 'O'),
 ('the', 'O'),
 ('capital', 'O'),
 ('and', 'O'),
 ('largest', 'O'),
 ('city', 'O'),
 ('of', 'O'),
 ('England', 'B-LOC'),
 ('and', 'O'),
 ('the', 'O'),
 ('United', 'B-LOC'),
 ('Kingdom', 'I-LOC'),
 ('.', 'O'),
 ('The', 'O'),
 ('city', 'O'),
 ('stands', 'O'),
 ('on', 'O'),
 ('the', 'O'),
 ('River', 'B-LOC'),
 ('Thames', 'I-LOC'),
 ('in', 'O'),
 ('the', 'O'),
 ('south', 'O'),
 ('-', 'O'),
 ('east', 'O'),
 ('of', 'O'),
 ('England', 'B-LOC'),
 (',', 'O'),
 ('at', 'O'),
 ('the', 'O'),
 ('head', 'O'),
 ('of', 'O'),
 ('its', 'O'),
 ('50', 'O'),
 ('-', 'O'),
 ('mile', 'O'),
 ('(', 'O'),
 ('80', 'O'),
 ('km', 'O'),
 (')', 'O'),
 ('estuary', 'O'),
 ('leading', 'O'),
 ('to', 'O'),
 ('the', 'O'),
 ('North', 'B-LOC'),
 ('Sea', 'I-LOC'),
 ('.', 'O')]
```



London is the capital and largest city of **England** and the **United** **Kingdom**. The city stands on the **River** **Thames** in the south-east of **England**, at the head of its 50-mile (80 km) estuary leading to the **North** **Sea**.

We can observe that the model under the hood performs quite well though it has mistakenly assigned **LOC** tag to a second **Lincoln** appearance (there exists a Lincoln city out there but such prediction is incorrect in the context provided).

Nevertheless, web application is deployed and available under https://ner-tensorflowjs.glitch.me/web-app/ner-web-app.html.