tweeter-ner-nlp

May 23, 2024

1 Twitter: NER NLP Bussiness Case By Ratnesh

2 Problem statement

Context: Twitter is a microblogging and social networking service on which users post and interact with messages known as “tweets”. Every second, on average, around 6,000 tweets are tweeted on Twitter, corresponding to over 350,000 tweets sent per minute, 500 million tweets per day. Twitter wants to automatically tag and analyze tweets for better understanding of the trends and topics without being dependent on the hashtags that the users use. Many users do not use hashtags or sometimes use wrong or mis-spelled tags, so they want to completely remove this problem and create a system of recognizing important content of the tweets.

Objective: You need to train a model that will be able to identify the various named entities.

3 Data Description

Dataset is annotated with 10 fine-grained NER categories: person, geo-location, company, facility, product,music artist, movie, sports team, tv show and other. Dataset was extracted from tweets and is structured in CoNLL format., in English language. Containing in Text file format.

The CoNLL format is a text file with one word per line with sentences separated by an empty line. The first word in a line should be the word and the last word should be the label.

Consider the two sentences below;

Harry Potter was a student living in london

Albus Dumbledore went to the Disney World

These two sentences can be prepared in a CoNLL formatted text file as follows.

1.Harry B-PER

2.Potter I-PER

3.was O

4.a O

5.student O

6.Living O

7.in O

8.London B-geo-loc

1.Albus B-PER

2.Dumbledore I-PER

3.went O

4.to O

5.the O

6.Disney B-facility

7.World I-facility

4 Process

1.Import the test, train data and understand the structure of the data.

1.1 usual exploratory analysis steps like checking the structure & characteristics of the dataset

2.Data preprocessing

2.1 Preparing the data in the format required by models

3.Training a LSTM + CRF Model

3.1 Use a word2vec model to initialize embeddings

4.Train a Tensorflow Tokenizer

5.Train test split the data

6.Train the model

6.1 Understand the implications of using bidirectional models

6.2 Try various hyperparameters to improve results

7.Align labels to input

7.1 Compute metrics to show effectiveness of trained model

8.Loading the Transformer model

8.1 You can use the 'bert-base-uncased' model from the transformers library

8.2 Load the model and the model config

9.Get the tokenizer for that model

10.Tokenize the data

10.1 Understand what change tokenization has done to the data

10.2 After tokenizing check if the data is correct and do we need to remove or add some tokens

11.Train test split the data

12.Train the model on the data

12.1 Try various hyperparameters

12.2 Try different training epochs, early stopping, various optimizer,metrics

13.Align the output for each sub tokens

13.1 Because we have result for sub tokens we need to combine back the tokens

14.Make some predictions and see if the trained models are working fine

14.1 Save the transformers model with model.save_pretrained method

14.2 Check the result on some of your own sentences

4.1 Downloading data

[1]:
```
!gdown 14_VHffl1qBUEnZ1IWFHnh6B9M5_A-Wf8
!gdown 1cnrGjppPOU_NtHNpGu0RJGg1CUNNsse_
```

```
Downloading…
From: https://drive.google.com/uc?id=14_VHffl1qBUEnZ1IWFHnh6B9M5_A-Wf8
To: /content/wnut 16.txt.conll
100% 403k/403k [00:00<00:00, 4.71MB/s]
Downloading…
From: https://drive.google.com/uc?id=1cnrGjppPOU_NtHNpGu0RJGg1CUNNsse_
To: /content/wnut 16test.txt.conll
100% 635k/635k [00:00<00:00, 6.20MB/s]
```

4.2 Installing libraries

[2]:
```
%pip install datasets transformers
%pip install tensorflow-addons
```

```
Collecting datasets
  Downloading datasets-2.16.1-py3-none-any.whl (507 kB)
                               507.1/507.1

kB 6.7 MB/s eta 0:00:00
Requirement already satisfied: transformers in
/usr/local/lib/python3.10/dist-packages (4.35.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from datasets) (3.13.1)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-
packages (from datasets) (1.23.5)
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dist-
packages (from datasets) (10.0.1)
Requirement already satisfied: pyarrow-hotfix in /usr/local/lib/python3.10/dist-
packages (from datasets) (0.6)
Collecting dill<0.3.8,>=0.3.0 (from datasets)
  Downloading dill-0.3.7-py3-none-any.whl (115 kB)
```

```
                           115.3/115.3

kB 11.5 MB/s eta 0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-
packages (from datasets) (1.5.3)
Requirement already satisfied: requests>=2.19.0 in
/usr/local/lib/python3.10/dist-packages (from datasets) (2.31.0)
Requirement already satisfied: tqdm>=4.62.1 in /usr/local/lib/python3.10/dist-
packages (from datasets) (4.66.1)
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packages
(from datasets) (3.4.1)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.16-py310-none-any.whl (134 kB)
                           134.8/134.8

kB 9.1 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]<=2023.10.0,>=2023.1.0 in
/usr/local/lib/python3.10/dist-packages (from datasets) (2023.6.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-
packages (from datasets) (3.9.1)
Requirement already satisfied: huggingface-hub>=0.19.4 in
/usr/local/lib/python3.10/dist-packages (from datasets) (0.20.3)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from datasets) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from datasets) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.1)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->datasets) (23.2.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (6.0.4)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->datasets) (1.9.4)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.4.1)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: async-timeout<5.0,>=4.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (4.0.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.19.4->datasets)
(4.5.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
```

```
/usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->datasets)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests>=2.19.0->datasets) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->datasets)
(2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->datasets)
(2023.11.17)
INFO: pip is looking at multiple versions of multiprocess to determine which
version is compatible with other requirements. This could take a while.
  Downloading multiprocess-0.70.15-py310-none-any.whl (134 kB)
                           134.8/134.8

kB 2.8 MB/s eta 0:00:00
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas->datasets) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.8.1->pandas->datasets) (1.16.0)
Installing collected packages: dill, multiprocess, datasets
Successfully installed datasets-2.16.1 dill-0.3.7 multiprocess-0.70.15
Collecting tensorflow-addons
  Downloading tensorflow_addons-0.23.0-cp310-cp310-manylinux_2_17_x86_64.manylin
ux2014_x86_64.whl (611 kB)
                           611.8/611.8

kB 5.0 MB/s eta 0:00:00
Requirement already satisfied: packaging in
/usr/local/lib/python3.10/dist-packages (from tensorflow-addons) (23.2)
Collecting typeguard<3.0.0,>=2.7 (from tensorflow-addons)
  Downloading typeguard-2.13.3-py3-none-any.whl (17 kB)
Installing collected packages: typeguard, tensorflow-addons
Successfully installed tensorflow-addons-0.23.0 typeguard-2.13.3
```

[3]:
```python
import pandas as pd
import tensorflow as tf
```

4.3 Loading data from the files

[4]:
```python
def load_data(filename: str):
  # Conll file is stored as (token, tag) pairs, one per line
  # Extracting data from conll files
  with open(filename, 'r') as file:
    lines = [line[:-1].split() for line in file] # Skipping last line as it␣
↪will be a blank space
```

```
  samples, start = [], 0
  for end, parts in enumerate(lines):
      if not parts:
          sample = [(token, tag)
                      for token, tag in lines[start:end]]
          samples.append(sample)
          start = end + 1
  if start < end:
    samples.append(lines[start:end])
  return samples

train_samples = load_data('wnut 16.txt.conll')
test_samples = load_data('wnut 16test.txt.conll')
samples = train_samples + test_samples
schema = ['_'] + sorted({tag for sentence in samples
                             for _, tag in sentence}) # '_' is used to indicate␣
 ↪a null (blank) token.
```

4.4 Structure of data

[5]:
```
train_samples[1]
```

[5]:
```
[('Made', 'O'),
 ('it', 'O'),
 ('back', 'O'),
 ('home', 'O'),
 ('to', 'O'),
 ('GA', 'B-geo-loc'),
 ('.', 'O'),
 ('It', 'O'),
 ('sucks', 'O'),
 ('not', 'O'),
 ('to', 'O'),
 ('be', 'O'),
 ('at', 'O'),
 ('Disney', 'B-facility'),
 ('world', 'I-facility'),
 (',', 'O'),
 ('but', 'O'),
 ('its', 'O'),
 ('good', 'O'),
 ('to', 'O'),
 ('be', 'O'),
 ('home', 'O'),
 ('.', 'O'),
 ('Time', 'O'),
 ('to', 'O'),
 ('start', 'O'),
 ('planning', 'O'),
 ('the', 'O'),
 ('next', 'O'),
 ('Disney', 'B-facility'),
 ('World', 'I-facility'),
 ('trip', 'O'),
 ('.', 'O')]
```

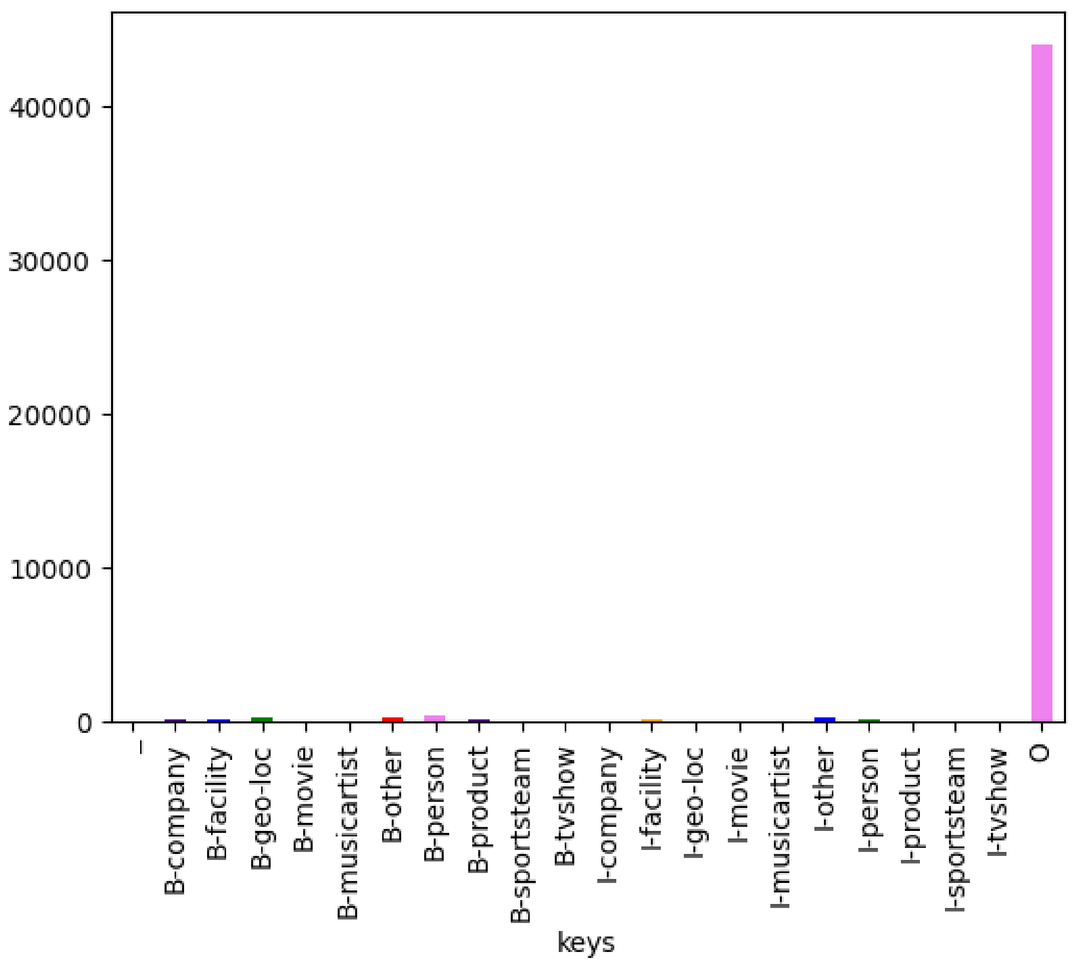
4.5 EDA: Let's have a look at the distribution of tags on data

[6]:
```
import seaborn as sns
colors = ['violet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']
counts = {}

# Calculateing the number of data points having a given label
for tag in schema:
  counts[tag] = 0
  for sample in train_samples:
    for label in sample:
      if label[1] == tag:
        counts[tag]+=1

counts_df = pd.DataFrame({'keys': list(counts.keys()), 'values': list(counts.
 ↪values())})
counts_df.plot.bar(x='keys', y='values', legend=False, color=colors)
```

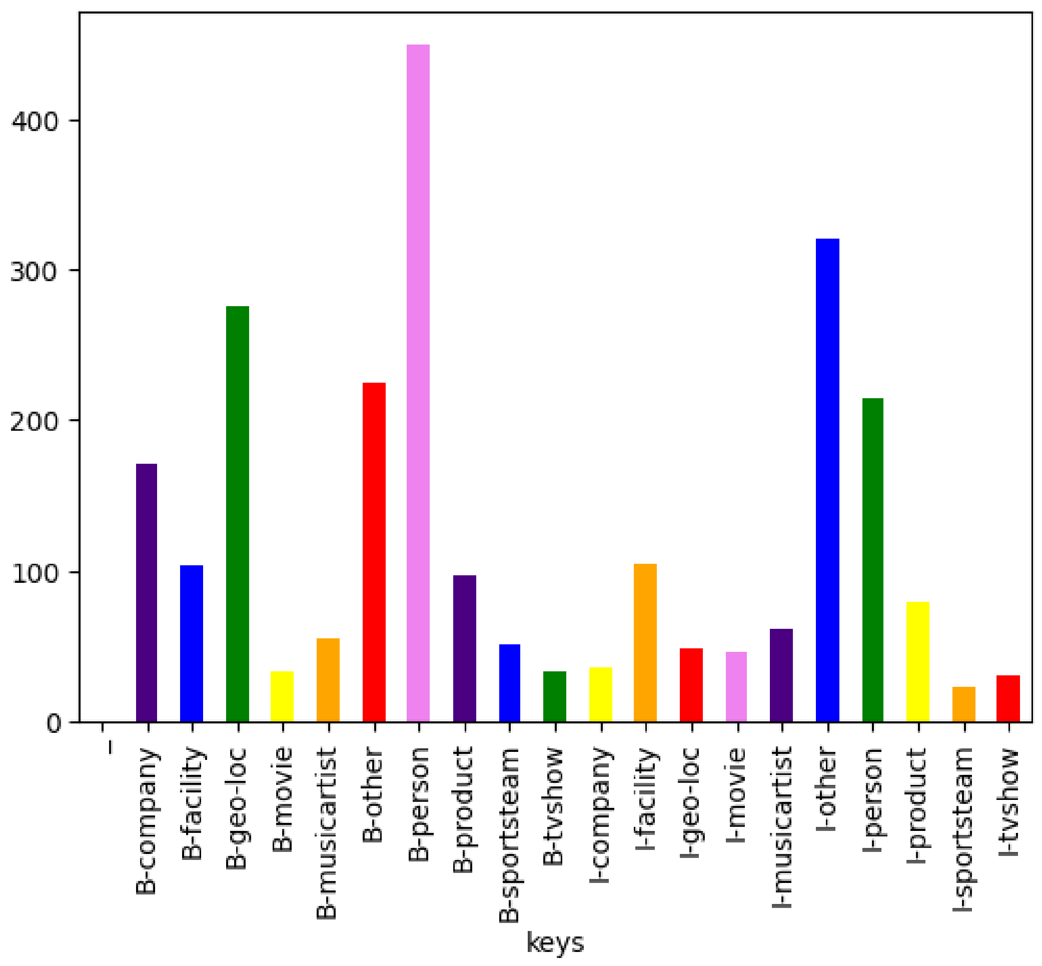
[6]:
```
<Axes: xlabel='keys'>
```



- We have too many "other" fields, which is natural as only few annotations exist per sentence
- let's remove O tag and see tag distribution

[7]:
```
counts.pop('O')
counts_df = pd.DataFrame({'keys': list(counts.keys()), 'values': list(counts.
 ↪values())})
counts_df.plot.bar(x='keys', y='values', legend=False, color=colors)
```

[7]:
```
<Axes: xlabel='keys'>
```



4.6 Tag information

- B-* Start token for a tag
- I-* Continuation tokens for a tag

4.7 Available Entities

- Company
- Facility
- Geo-loc: geolocation
- Musicartist
- Person
- Product
- Sportsteam
- TV Show
- Other

4.7.1 More preprocessing

- let's get vocab & sequence lengths

[8]:
```
from collections import defaultdict
all_samples = train_samples
all_samples.extend(test_samples)

word_counts = defaultdict(int) # Calculate vocab size
max_len = 0 # Calculate max length of a sentence

for sample in all_samples:
  for word in sample:
    word_counts[word[0]]+=1

  max_len = max(max_len, len(sample))

n_words = len(word_counts.items())


print("*"*30)
print("Max Length: ", max_len)
print("Vocab Size: ", n_words)
```

```
******************************
Max Length:  39
Vocab Size:  25382
```

4.8 Our approach

- Train a simple LSTM + CRF model to get a baseline
- Look at the results of transformer based architectures

4.9 Training LSTM + CRF model:

- Let's using glove to initialize embeddings

[9]:
```
import gensim.downloader as api
word2vec = api.load("glove-twitter-200") # Loading glove-twitter model
embedding_dim = 200
```

```
[==================================================] 100.0% 758.5/758.5MB
downloaded
```

4.9.1 Training a tokenizer for LSTM input embeddings

[10]:
```
all_sentences = [] # Concating test, train sentences. To train a tokenizer
for sample in all_samples:
  sentence = [tag[0] for tag in sample]
  all_sentences.append(sentence)

crf_tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=n_words,␣
 ↪lower=True)
crf_tokenizer.fit_on_texts(all_sentences)
```

4.9.2 Prepare embedding matrix

[11]:
```
import numpy as np
num_tokens = len(crf_tokenizer.word_index) + 1
hits = 0
misses = 0
missed_words = []


# Prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in crf_tokenizer.word_index.items():
  embedding_vector = None
  try:
    embedding_vector = word2vec[word]
  except Exception :
    pass

  if embedding_vector is not None:
    # Words not found in embedding index will be all-zeros.
    # This includes the representation for "padding" and "OOV"
    embedding_matrix[i] = embedding_vector
    hits += 1
  else:
    missed_words.append(word)
    misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 11495 words (10438 misses)
```

5 LSTM + CRF Model training

5.1 Creating a training dataset

[12]:
```
tag2id = {} # Label to indicies mapping
id2tag = {} # Index to label mapping
for i, tag in enumerate(schema):
  tag2id[tag] = i
  id2tag[i] = tag
```

- We will encode our labels as OHE vectors. This is to keep it compatible with SigmoidFocal-CrossEntropy loss

[13]:
```python
def get_dataset(samples, max_len, tag2id, tokenizer):
  '''Prepares the input dataset

  Args:
    `samples`: List[List[Tuple[word, tag]]], input data
    `max_len`: Maximum input length
    `tag2id`: Mapping[tag: integer]
    `tokenizer`: Tensorflow tokenizer, for tokenizing input sequence

  Returns:
    Tuple[np.ndarray, np.ndarray]: sentences and it's labels
  '''
  dataset = {'samples':[], 'labels': []}

  for sample in samples:
    # Extracting inputs and labels
    inputs = [x[0] for x in sample]
    outputs = [x[1] for x in sample]

    # Tokenizing inputs
    inputs = tokenizer.texts_to_sequences([inputs])[0]

    # padding labels
    padded_inputs = [inputs[i] if i < len(inputs) else 0 for i in␣
 ↪range(max_len)]

    # Initializing labels as One Hot Encoded Vectors
    padded_labels = [[0 for i in range(len(tag2id))] for j in range(max_len)]
    for i in range(len(outputs)):
      padded_labels[i][tag2id[outputs[i]]] = 1

    # Adding padded inputs & labels to dataset
    dataset['samples'].append(padded_inputs)
    dataset['labels'].append(padded_labels)

  return np.array(dataset['samples']), np.array(dataset['labels'])

train_sentences, train_labels = get_dataset(train_samples, max_len, tag2id,␣
 ↪crf_tokenizer)
test_sentences, test_labels = get_dataset(test_samples, max_len, tag2id,␣
 ↪crf_tokenizer)
```

5.2 Training Model

- using sigmoid focal cross entropy loss. It performs better than sparse categorical cross entropy for highly imbalanced data.

[14]:
```
from keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow_addons.utils.types import FloatTensorLike, TensorLike

# LSTM components
from keras.layers import LSTM, Embedding, Dense, TimeDistributed, Dropout,␣
 ↪Bidirectional

# CRF layer
from tensorflow_addons.layers import CRF

# Sigmoid focal cross entropy loss. works well with highly unbalanced input data
from tensorflow_addons.losses import SigmoidFocalCrossEntropy
from tensorflow_addons.optimizers import AdamW


def build_model():
  # Model definition
  input = Input(shape=(max_len,))

  # Get embeddings
  embeddings = Embedding(input_dim=embedding_matrix.shape[0],
                      output_dim=embedding_dim,
                      input_length=max_len, mask_zero=True,
                      embeddings_initializer=tf.keras.initializers.
 ↪Constant(embedding_matrix)
                    )(input)

  # variational biLSTM
  output_sequences = Bidirectional(LSTM(units=50,␣
 ↪return_sequences=True))(embeddings)

  # Stacking
  output_sequences = Bidirectional(LSTM(units=50,␣
 ↪return_sequences=True))(output_sequences)

  # Adding more non-linearity
  dense_out = TimeDistributed(Dense(25, activation="relu"))(output_sequences)

  # CRF layer
  crf = CRF(len(schema), name='crf')
  predicted_sequence, potentials, sequence_length, crf_kernel = crf(dense_out)
  model = Model(input, potentials)
  model.compile(
      optimizer=AdamW(weight_decay=0.001),
      loss= SigmoidFocalCrossEntropy()) # Sigmoid focal cross entropy loss

  return model

model = build_model()

# Checkpointing
save_model = tf.keras.callbacks.ModelCheckpoint(filepath='twitter_ner_crf.h5',
  monitor='val_loss',
  save_weights_only=True,
  save_best_only=True,
  verbose=1
)

# Early stoppings
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', verbose=1, patience=1)

callbacks = [save_model, es]

model.summary()
```

```
/usr/local/lib/python3.10/dist-
packages/tensorflow_addons/utils/tfa_eol_msg.py:23: UserWarning:

TensorFlow Addons (TFA) has ended development and introduction of new features.
TFA has entered a minimal maintenance and release mode until a planned end of
life in May 2024.
Please modify downstream libraries to take dependencies from other repositories
in our TensorFlow community (e.g. Keras, Keras-CV, and Keras-NLP).

For more information see: https://github.com/tensorflow/addons/issues/2807

  warnings.warn(
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 39)]              0

 embedding (Embedding)       (None, 39, 200)           4386800

 bidirectional (Bidirection  (None, 39, 100)           100400
 al)
 bidirectional_1 (Bidirecti  (None, 39, 100)           60400
 onal)

 time_distributed (TimeDist  (None, 39, 25)            2525
 ributed)

 crf (CRF)                   [(None, 39),              1100
                              (None, 39, 22),
                              (None,),
                              (22, 22)]

=================================================================
Total params: 4551225 (17.36 MB)
Trainable params: 4551225 (17.36 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

6 Training our model

[15]:
```
model.fit(train_sentences, train_labels,
          validation_data = (test_sentences, test_labels),
          epochs = 300,
          callbacks  = callbacks,
          shuffle=True)
```

```
Epoch 1/300

WARNING:tensorflow:Gradients do not exist for variables ['chain_kernel:0'] when
minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss` argument?
WARNING:tensorflow:Gradients do not exist for variables ['chain_kernel:0'] when
minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss` argument?

194/196 [============================>.] - ETA: 0s - loss: 0.0850
Epoch 1: val_loss improved from inf to 0.04047, saving model to
twitter_ner_crf.h5
196/196 [==============================] - 35s 60ms/step - loss: 0.0847 -
val_loss: 0.0405
Epoch 2/300
194/196 [============================>.] - ETA: 0s - loss: 0.0353
Epoch 2: val_loss improved from 0.04047 to 0.03353, saving model to
twitter_ner_crf.h5
196/196 [==============================] - 6s 31ms/step - loss: 0.0353 -
val_loss: 0.0335
Epoch 3/300
196/196 [==============================] - ETA: 0s - loss: 0.0266
Epoch 3: val_loss improved from 0.03353 to 0.02444, saving model to
```

```
twitter_ner_crf.h5
196/196 [==============================] - 9s 48ms/step - loss: 0.0266 -
val_loss: 0.0244
Epoch 4/300
194/196 [============================>.] - ETA: 0s - loss: 0.0213
Epoch 4: val_loss improved from 0.02444 to 0.01983, saving model to
twitter_ner_crf.h5
196/196 [==============================] - 7s 36ms/step - loss: 0.0213 -
val_loss: 0.0198
Epoch 5/300
194/196 [============================>.] - ETA: 0s - loss: 0.0172
Epoch 5: val_loss improved from 0.01983 to 0.01764, saving model to
twitter_ner_crf.h5
196/196 [==============================] - 6s 30ms/step - loss: 0.0172 -
val_loss: 0.0176
Epoch 6/300
194/196 [============================>.] - ETA: 0s - loss: 0.0201
Epoch 6: val_loss did not improve from 0.01764
196/196 [==============================] - 5s 26ms/step - loss: 0.0202 -
val_loss: 0.0185
Epoch 6: early stopping
```

[15]:
```
<keras.src.callbacks.History at 0x7b8c72e9a950>
```

7 Let's load the best model

[16]:
```
model.load_weights('twitter_ner_crf.h5')
```

[17]:
```
crf_model = tf.keras.Model(inputs=model.input, outputs=[model.output, model.
 ↪get_layer('crf').output, model.input])
```

7.0.1 Let's calculate average accuracy of the model on test set

[18]:
```
def calculate_accuracy(y_true, y_pred):
    '''Convert categorical one hot encodings to indices and compute accuracy

    Args:
      `y_true`: true values
      `y_pred`: model predictions

    Returns:
      Integer, accuracy of prediction
    '''
    acc_metric = tf.keras.metrics.Accuracy()
    y_true = tf.argmax(y_true, axis=-1)
    return acc_metric(y_true, y_pred).numpy().item()
```

```
def calculate_mosacy(crf_model, test_sentences, test_labels):
  '''Calculates average validation accuracy of model'''

  # Batch the dataset
  batched_validation_set = tf.data.Dataset.from_tensor_slices((test_sentences,␣
 ↪test_labels)).batch(32)

  average_acc = 0
  # Iterate through batches
  for batch_test_sentences, batch_test_labels in batched_validation_set:
    predicted_labels, _, _, _ = crf_model(batch_test_sentences)[1]
    average_acc += calculate_accuracy(batch_test_labels, predicted_labels)

  average_acc/=len(batched_validation_set)
  return average_acc

average_acc = calculate_mosacy(crf_model, test_sentences, test_labels)

print("*"*32)
print(f"Average accuracy of model on test set: {average_acc:.3f}")
```

```
********************************
Average accuracy of model on test set: 0.961
```

8 BERT Model

8.1 Getting the bert model

[19]:
```
from transformers import AutoConfig, TFAutoModelForTokenClassification

MODEL_NAME = 'bert-base-uncased'
```

9 Loading the tokenizer

[20]:
```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME) # Load bert-base-uncased␣
 ↪tokenizer
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
```

```
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(

tokenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]

config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]

vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

- tokenizer adds 101 and 102 token id at the start and end of the tokens
- using[1:-1] to eliminate the extra 101, 102 that tokenizer adds
- Let us have a peak at tokenization of a training sample

[21]:
```python
sample=train_samples[10] # Random tokenized sample
for token, tag in sample:
  for subtoken in tokenizer(token)['input_ids'][1:-1]:
    print(token,subtoken)
```

```
RT 19387
@Hatshepsutely 1030
@Hatshepsutely 16717
@Hatshepsutely 5369
@Hatshepsutely 4523
@Hatshepsutely 10421
@Hatshepsutely 2135
: 1024
@adamlambert 1030
@adamlambert 4205
@adamlambert 10278
@adamlambert 8296
please 3531
, 1010
oh 2821
please 3531
wear 4929
the 1996
infamous 14429
beach 3509
hat 6045
tonight 3892
during 2076
your 2115
encore 19493
( 1006
in 1999
lieu 22470
of 1997
a 1037
rasta 20710
rasta 2696
wig) 24405
wig) 1007
. 1012
&lt; 1004
&lt; 8318
&lt; 1025
3333 21211
3333 2509
```

9.0.1 Get Datasets

[22]:
```python
import numpy as np
import tqdm

def tokenize_sample(sample):
  # Expand label to all subtokens and add 'O' label to start and end tokens
  seq = [
    (subtoken, tag)
    for token, tag in sample
    for subtoken in tokenizer(token.lower())['input_ids'][1:-1]
  ]
  return [(3, 'O')] + seq + [(4, 'O')]

def preprocess(samples, tag2id):
  tokenized_samples = list((map(tokenize_sample, samples)))
  max_len = max(map(len, tokenized_samples))

  # Subtokens
  X_input_ids = np.zeros((len(samples), max_len), dtype=np.int32)

  # Masks
  X_input_masks = np.zeros((len(samples), max_len), dtype=np.int32)

  # labels
  y = np.zeros((len(samples), max_len), dtype=np.int32)

  for i, sentence in enumerate(tokenized_samples):
    for j in range(len(sentence)):
      X_input_masks[i, j] = 1
    for j, (subtoken_id, tag) in enumerate(sentence):
      X_input_ids[i, j] = subtoken_id
      y[i, j] = tag2id[tag]
  return (X_input_ids, X_input_masks), y
```

```
X_train, y_train = preprocess(train_samples, tag2id)
X_test, y_test = preprocess(test_samples, tag2id)
```

9.1 Loading model

[23]:
```
config = AutoConfig.from_pretrained(MODEL_NAME, num_labels=len(schema),
                                    id2tag=id2tag, tag2id=tag2id) # Bert config

model = TFAutoModelForTokenClassification.from_pretrained(MODEL_NAME,
                                                          config=config) #␣
 ↪Loading Bert model
model.summary()
```

```
model.safetensors:   0%|          | 0.00/440M [00:00<?, ?B/s]

All PyTorch model weights were used when initializing
TFBertForTokenClassification.

Some weights or buffers of the TF 2.0 model TFBertForTokenClassification were
not initialized from the PyTorch model and are newly initialized:
['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.

Model: "tf_bert_for_token_classification"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bert (TFBertMainLayer)      multiple                  108891648

 dropout_37 (Dropout)        multiple                  0

 classifier (Dense)          multiple                  16918

=================================================================
Total params: 108908566 (415.45 MB)
Trainable params: 108908566 (415.45 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

9.1.1 Fit model on training data

[24]:
```
BATCH_SIZE=32

optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001) # Creating optimizer

loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
```

```
model.compile(optimizer=optimizer, loss=loss, metrics=metric)

history = model.fit(X_train, y_train,
                    validation_split=0.2, epochs=10,
                    batch_size=BATCH_SIZE)
```

```
Epoch 1/10
157/157 [==============================] - 221s 1s/step - loss: 0.2322 -
accuracy: 0.9436 - val_loss: 0.0961 - val_accuracy: 0.9845
Epoch 2/10
157/157 [==============================] - 172s 1s/step - loss: 0.0576 -
accuracy: 0.9882 - val_loss: 0.0557 - val_accuracy: 0.9862
Epoch 3/10
157/157 [==============================] - 173s 1s/step - loss: 0.0372 -
accuracy: 0.9906 - val_loss: 0.0482 - val_accuracy: 0.9871
Epoch 4/10
157/157 [==============================] - 173s 1s/step - loss: 0.0239 -
accuracy: 0.9936 - val_loss: 0.0433 - val_accuracy: 0.9892
Epoch 5/10
157/157 [==============================] - 174s 1s/step - loss: 0.0157 -
accuracy: 0.9959 - val_loss: 0.0395 - val_accuracy: 0.9901
Epoch 6/10
157/157 [==============================] - 174s 1s/step - loss: 0.0103 -
accuracy: 0.9975 - val_loss: 0.0462 - val_accuracy: 0.9902
Epoch 7/10
157/157 [==============================] - 175s 1s/step - loss: 0.0076 -
accuracy: 0.9981 - val_loss: 0.0454 - val_accuracy: 0.9909
Epoch 8/10
157/157 [==============================] - 175s 1s/step - loss: 0.0076 -
accuracy: 0.9981 - val_loss: 0.0490 - val_accuracy: 0.9915
Epoch 9/10
157/157 [==============================] - 175s 1s/step - loss: 0.0047 -
accuracy: 0.9988 - val_loss: 0.0497 - val_accuracy: 0.9908
Epoch 10/10
157/157 [==============================] - 175s 1s/step - loss: 0.0049 -
accuracy: 0.9988 - val_loss: 0.0506 - val_accuracy: 0.9914
```

- Lets have a side by side view of true labels and model predictions
- Arranged as an array of Tuple(token, true label, model prediction)

[25]:
```
def aggregate(sample, predictions):
    results = []
    i = 1
    for token, y_true in sample:
        nr_subtoken = len(tokenizer(token.lower())['input_ids']) - 2 # Extracting␣
 ↪word tokens
        pred = predictions[i:i+nr_subtoken] # Extracting predictions
        i += nr_subtoken
        y_pred = schema[np.argmax(np.sum(pred, axis=0))] # Get label of prediction
        results.append((token, y_true, y_pred))
    return results

y_probs = model.predict(X_test)[0]
predictions = [aggregate(sample, predictions)
               for sample, predictions in zip(test_samples, y_probs)]
```

```
121/121 [==============================] - 43s 333ms/step
```

[26]:
```
for i in range(10,15):
    print(predictions[i])
```

```
[('I', 'O', 'O'), ('drive', 'O', 'O'), ('by', 'O', 'O'), ('that', 'O', 'O'),
('motel', 'O', 'O'), ('almost', 'O', 'O'), ('every', 'O', 'O'), ('night', 'O',
'O'), ('.', 'O', 'O'), ('#MesaShooting', 'O', 'O')]
[('Apple', 'B-product', 'B-product'), ('MacBook', 'I-product', 'I-product'),
('Pro', 'I-product', 'I-product'), ('A1278', 'I-product', 'I-product'), ('13.3',
'I-product', 'I-product'), ('"', 'I-product', 'I-product'), ('Laptop',
'I-product', 'I-product'), ('-', 'I-product', 'I-product'), ('MD101LL/A',
'I-product', 'I-product'), ('(', 'O', 'O'), ('June', 'O', 'O'), (',', 'O', 'O'),
('2012', 'O', 'O'), (')', 'O', 'O'), ('-', 'O', 'O'), ('Full', 'O', 'O'),
('read', 'O', 'O'), ('by', 'O', 'O'), ('eBay', 'B-company', 'B-company'),
('http://t.co/2zgQ99nmuf', 'O', 'O'), ('http://t.co/eQmogqqABK', 'O', 'O')]
[('Tuff', 'B-musicartist', 'B-musicartist'), ('Culture', 'I-musicartist',
'I-musicartist'), ('-', 'O', 'O'), ('Destiny', 'B-product', 'B-product'), ('EP',
'O', 'O'), ('(', 'O', 'O'), ('PAR', 'O', 'O'), ('042', 'O', 'O'),
('FORTHCOMING', 'O', 'O'), ('27th', 'O', 'O'), ('JULY', 'O', 'O'), ('VIA', 'O',
'O'), ('JUNO', 'B-product', 'B-product'), (')', 'O', 'O'), ('Tracklist', 'O',
'O'), (':', 'O', 'O'), ('Destiny', 'B-product', 'B-product'), ('Questions',
'B-product', 'B-product'), ('Theres', 'B-product', 'B-product'), ('No',
'I-product', 'B-product'), ('…', 'O', 'O'), ('http://t.co/X7nL8DiREK', 'O',
'O')]
[('December', 'O', 'O'), ('23', 'O', 'O'), (',', 'O', 'O'), ('2015', 'O', 'O'),
('at', 'O', 'O'), ('03:44', 'O', 'O'), ('PM', 'O', 'O'), ('#if24', 'O', 'O'),
('#s8', 'O', 'O')]
[('RT', 'O', 'O'), ('@YahooDrSaturday', 'O', 'O'), (':', 'O', 'O'), ('This',
'O', 'O'), ('is', 'O', 'O'), ('how', 'O', 'O'), ('Arkansas', 'B-sportsteam',
'B-sportsteam'), ('crazily', 'O', 'O'), ('converted', 'O', 'O'), ('4th', 'O',
'O'), ('and', 'O', 'O'), ('25', 'O', 'O'), ('in', 'O', 'O'), ('OT', 'O', 'O'),
('.', 'O', 'O'), ('What', 'O', 'O'), ('a', 'O', 'O'), ('lateral', 'O', 'O'),
('!', 'O', 'O'), ('https://t.co/ylALEACWe8', 'O', 'O')]
```

[27]:
```
model.save_pretrained("output/NER_pretrained")
```

10 Comparision

[28]:
```
def tokenize_bert(sentence):
  sentence_tokens = tokenizer(sentence.split(' '))['input_ids'] # Splitting␣
 ↪sentence into word tokens
  ner_tokens = [3] # Start token
  for word_token in sentence_tokens:
    ner_tokens.extend(word_token[1:-1]) # Adding tokenized word token indicies
  ner_tokens += [4] # End token
  return ner_tokens
```

[29]:
```
sentence ="apple macbook pro is the best laptop in the world"

# Bert tokenization
bert_tokens = tokenize_bert(sentence)

# CRF tokenization
crf_tokens = crf_tokenizer.texts_to_sequences([sentence])
```

Bert Output

[30]:
```
def align_labels_to_input(sentence, predictions):
  sentence_tokens = sentence.lower().split(" ")
  results = []

  i = 1
  # Extracting labels corresponding to tokens
  for token in sentence_tokens:
      nr_subtoken = len(tokenizer(token)['input_ids']) - 2
      pred = predictions[i:i+nr_subtoken]
      i += nr_subtoken
      y_pred = id2tag[np.argmax(np.sum(pred, axis=0))]
      results.append((token, y_pred))
  return results


bert_logits = model.predict([bert_tokens], verbose=0).logits

align_labels_to_input(sentence, bert_logits[0])
```

[30]:
```
[('apple', 'B-product'),
 ('macbook', 'I-product'),
 ('pro', 'I-product'),
 ('is', 'O'),
 ('the', 'O'),
 ('best', 'O'),
 ('laptop', 'O'),
 ('in', 'O'),
 ('the', 'O'),
 ('world', 'O')]
```

CRF Output

[31]:
```python
from pprint import pprint # Pretty print package

crf_padded_tokens = [[crf_tokens[0][x] if x < len(crf_tokens[0]) else 0 for x␣
 ↪in range(39)]]
crf_preds, _, _, _ = crf_model.predict(crf_padded_tokens, verbose=0)[1]

crf_preds = [id2tag[x] for x in crf_preds[0]] # Convert indicies into␣
 ↪predictions


# Get aligned inputs with labels
input_word_tokens = [crf_tokenizer.sequences_to_texts([[x]])[0] for x in␣
 ↪crf_padded_tokens[0]]

# Only printing non-padded tokens with their labels
pprint(list(zip(input_word_tokens[:len(crf_tokens[0])], crf_preds[:
 ↪len(crf_tokens[0])])))
```

```
[('apple', 'B-other'),
 ('macbook', 'I-tvshow'),
 ('pro', 'O'),
 ('is', 'O'),
 ('the', 'O'),
 ('best', 'O'),
 ('laptop', 'O'),
 ('in', 'O'),
 ('the', 'O'),
 ('world', 'O')]
```

[31]:
```
```

11 Questions

#1 Defining the problem statements and where can this and modifications of this be used?

Problem Statement A problem statement is a clear and concise description of the issue to be addressed. It includes:

Problem Description: A specific explanation of the problem. Context and Background: Information on why the problem exists. Impact: Consequences of the problem. Desired Outcome: What successful resolution looks like.

#2 Explain the data format (conll bio format) ?

The CoNLL BIO format is a commonly used data format for annotating text in tasks like Named Entity Recognition (NER). “BIO” stands for “Beginning”, “Inside”, and “Outside”, which are the tags used to denote the start of an entity, tokens inside an entity, and tokens outside any entity, respectively. Here’s a detailed explanation of the format:

Structure of CoNLL BIO Format

Tokens: Each word in the text is treated as a token and annotated separately. Tags: Each token is assigned a tag indicating whether it is at the beginning (B), inside (I), or outside (O) of a named entity. The type of entity (e.g., PERSON, LOCATION, ORGANIZATION) is also specified.

Example Consider the sentence: “John Doe is a doctor in New York City.”

The CoNLL BIO format annotation might look like this:

mathematica Copy code John B-PER Doe I-PER is O a O doctor O in O New B-LOC York I-LOC City I-LOC

Explanation B-PER: Indicates the beginning of a person entity (“John”). I-PER: Indicates the inside of a person entity (“Doe”). O: Indicates that the token is outside any named entity (“is”, “a”, “doctor”, “in”). B-LOC: Indicates the beginning of a location entity (“New”). I-LOC: Indicates the inside of a location entity (“York”, “City”).

Characteristics

Token-level Annotation: Each token (word) is annotated individually. Entity Boundaries: “B-” marks the beginning of an entity, “I-” marks the continuation, and “O” marks tokens outside any entity. Entity Types: The tags are suffixed with the entity type (e.g., PER for person, LOC for location). Benefits Clarity: Clearly distinguishes between the start and continuation of entities. Simplicity: Straightforward format that is easy to implement and understand. Flexibility: Can be used for various types of entities by simply changing the suffix (e.g., B-ORG for organizations).

Use Cases

NER Training: Commonly used for training NER models. Shared Tasks: Frequently used in shared tasks and competitions, such as the CoNLL-2003 NER task.

#3 What other ner data annotation formats are available and how are they different ?

Several Named Entity Recognition (NER) data annotation formats exist, each with its own structure:

IOB (Inside-Outside-Beginning): B-PER for the beginning of a person entity, I-PER for inside, and O for outside any entity. IOB2: Similar to IOB, but every entity starts with a B- regardless of whether it immediately follows another entity of the same type. IO: I- for all tokens inside an entity, O for outside. BILOU (Beginning-Inside-Last-Outside-Unit): B- for beginning, I- for inside, L- for last token in a multi-token entity, O for outside, and U- for unit (single-token entity). Each format varies in how it marks entity boundaries, affecting model complexity and performance.

#4 Why do we need tokenization of the data in our case ?

Tokenization is a crucial preprocessing step in Natural Language Processing (NLP) tasks, including Named Entity Recognition (NER), because it breaks down text into manageable pieces, usually words or subwords. Here’s why tokenization is essential:

Consistent Input Format: Tokenization standardizes the text into a format that models can understand. NLP models require text to be divided into tokens to process and analyze the text systematically.

Handling Different Lengths: Real-world text varies in length and complexity. Tokenization ensures that longer texts are divided into smaller, more manageable units, making it easier for models to handle input of varying lengths.

Capturing Meaning: Words or subwords are the fundamental units of meaning. By tokenizing text, models can focus on these meaningful units, allowing for more accurate analysis and understanding.

Model Compatibility: Advanced models like BERT and its variants expect tokenized input. Tokenizers tailored for these models often include special handling of out-of-vocabulary words and subword segmentation, which helps in managing rare or complex words effectively.

Efficient Processing: Tokenization allows for more efficient and parallel processing of text data, enabling models to learn from and process large datasets quickly.

#5.What other models can you use for this task?

For the task of Named Entity Recognition (NER), there are several models and approaches you can use besides BERT. Here are some alternatives, along with their strengths and typical use cases:

Traditional Machine Learning Models Conditional Random Fields (CRF): Strengths: CRFs are effective for sequence labeling tasks because they consider the context of the entire sequence. Use Cases: Suitable for smaller datasets and when computational resources are limited. Example: A CRF model might be used for NER in a specific domain like biomedical text where annotated data is limited. Neural Network Models BiLSTM-CRF (Bidirectional Long Short-Term Memory - Conditional Random Fields):

Strengths: Combines the advantages of BiLSTM for capturing long-range dependencies and CRF for structured prediction. Use Cases: Commonly used in many NER systems for its effectiveness in capturing context from both directions. Example: A BiLSTM-CRF model can be used in real-time chatbots to recognize user intent and entities from user inputs. BiLSTM:

Strengths: Capable of handling sequence data and capturing dependencies from both past and future contexts. Use Cases: Suitable for tasks where long-term dependencies in the text are crucial. Example: A BiLSTM can be used for tagging parts of speech or extracting entities from longer paragraphs of text. Transformer-Based Models RoBERTa (Robustly optimized BERT approach):

Strengths: An optimized version of BERT, it performs better on many NLP tasks due to improved training techniques and more data. Use Cases: Suitable for tasks requiring high accuracy and robustness. Example: RoBERTa can be used in financial document analysis to accurately extract entities like company names, financial figures, etc. DistilBERT:

Strengths: A smaller, faster, and lighter version of BERT, designed to retain 97% of BERT's performance while being more efficient. Use Cases: Suitable for scenarios with limited computational resources. Example: DistilBERT can be used in mobile applications where real-time NER is needed. GPT-3 (Generative Pre-trained Transformer 3):

Strengths: Known for its large-scale capacity and ability to perform a variety of NLP tasks with few-shot learning. Use Cases: Suitable for complex tasks requiring nuanced understanding and generation of text. Example: GPT-3 can be used for conversational agents that need to understand and generate responses based on user-provided entities. ALBERT (A Lite BERT):

Strengths: A lighter and faster version of BERT, designed to reduce memory consumption and increase training speed. Use Cases: Suitable for environments with limited resources but requiring robust performance. Example: ALBERT can be used in academic research settings where computational resources are limited. Other Neural Models SpaCy:

Strengths: An open-source library with pre-trained NER models that are efficient and easy to use. Use Cases: Suitable for quick implementation and deployment of NER systems. Example: SpaCy can be used in web applications to extract named entities from user-generated content in real-time. AllenNLP:

Strengths: Provides a wide range of pre-trained models and tools specifically for NLP tasks. Use Cases: Suitable for research and applications that require flexibility and state-of-the-art performance. Example: AllenNLP can be used in academic projects for comparative studies of different NER models.

#6.Did early stopping have any effect on the training and results?

Early stopping is a regularization technique used during training machine learning models to prevent overfitting. It monitors the model's performance on a validation set and stops training when the performance stops improving, which helps to achieve better generalization to new, unseen data.

Effects of Early Stopping on Training and Results Prevents Overfitting:

Without early stopping, a model might continue to learn patterns that are specific to the training data, leading to overfitting. Early stopping halts training before this happens, ensuring that the model captures the underlying patterns that generalize well to new data. Improves Generalization:

By stopping the training process at the point where the validation performance ceases to improve, early stopping helps in obtaining a model that performs better on the validation set and, by extension, on unseen test data. Reduces Training Time:

Early stopping can significantly cut down on training time by avoiding unnecessary epochs once the model has reached its optimal performance. This efficiency is particularly useful when dealing with large datasets and complex models. Optimal Model Selection:

It helps in selecting the model that is trained just enough to capture the most relevant features without overcomplicating the model structure. How Early Stopping Works Monitor Metric:

Early stopping requires a metric to monitor, typically the validation loss or validation accuracy. Patience Parameter:

The patience parameter defines how many epochs to wait after the last improvement before stopping the training. For example, if patience is set to 10, training will stop if there is no improvement in the validation metric for 10 consecutive epochs. Restore Best Weights:

Often, early stopping implementations restore the weights of the model from the epoch with the best validation performance, ensuring that the best model is used for inference. Example Scenario Consider training a neural network for text classification:

Without Early Stopping: The model might continue to train for 50 epochs. The training accuracy could keep increasing, while the validation accuracy starts to decline after 30 epochs, indicating overfitting. With Early Stopping: Training stops automatically at around 30 epochs when the validation accuracy stops improving, resulting in a model that performs better on unseen data.

Practical Implications In practical machine learning tasks, including those involving BERT or other transformers, early stopping is particularly valuable due to the following reasons:

Complex Models: Transformers like BERT have a large number of parameters and can easily overfit, making early stopping an effective tool to mitigate this risk. Resource Management: Training large models is resource-intensive. Early stopping helps in saving computational resources by avoiding unnecessary epochs.

#7.How does the BERT model expect a pair of sentences to be processed?

BERT (Bidirectional Encoder Representations from Transformers) processes a pair of sentences using a specific input format that allows it to understand the relationship between the sentences. Here's how BERT expects a pair of sentences to be processed:

Input Format for Sentence Pairs Special Tokens:

Token Sequence:

The input consists of the sequence: CLS Sentence A SEP Sentence B SEP. Token IDs:

Each token in the sentences is converted into its corresponding token ID from BERT's vocabulary. Segment IDs:

Segment A: All tokens in the first sentence (Sentence A) are assigned a segment ID of 0. Segment B: All tokens in the second sentence (Sentence B) are assigned a segment ID of 1. This helps BERT distinguish between the two sentences during processing. Attention Mask:

An attention mask is used to indicate which tokens should be attended to (1) and which should be ignored (0), typically for padding tokens. Example Consider the following pair of sentences:

Sentence A: "How are you?" Sentence B: "I am fine." The processing steps are as follows:

Tokenization:

Tokenize each sentence using BERT's tokenizer. Example tokens: ['CLS', 'How', 'are', 'you', '?', 'SEP', 'I', 'am', 'fine', '.', 'SEP'] Token IDs:

Convert tokens to their respective token IDs. Example token IDs: [101, 2129, 2024, 2017, 1029, 102, 1045, 2572, 2986, 1012, 102] Segment IDs:

Assign segment IDs: [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1] Attention Mask:

Example attention mask: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

#8.Why Choose Attention-Based Models over Recurrent-Based Models?

Parallelization: Attention models, like Transformers, allow for parallel processing, making training faster. Long-Range Dependencies: Attention mechanisms can capture long-range dependencies more effectively than RNNs or LSTMs. Scalability: Transformers scale better with larger datasets and model sizes.

#9.Differentiate BERT and Simple Transformers

BERT (Bidirectional Encoder Representations from Transformers):

Uses a transformer architecture. Pre-trained on large corpora using masked language modeling and next sentence prediction. Requires fine-tuning for specific tasks.

Simple Transformers:

A library built on top of the Hugging Face Transformers library. Simplifies the process of training and deploying transformer models. Provides easy-to-use interfaces for tasks like text classification, NER, and more.

In summary, defining clear problem statements is essential for effective problem-solving across various domains. The CoNLL BIO format is one of several annotation formats used in NER tasks, with tokenization playing a crucial role. Various models can be used for NER, and early stopping is a useful technique to improve model generalizability. Attention-based models like BERT offer significant advantages over recurrent models, and libraries like Simple Transformers make working with these models more accessible.