i. Define the Problem Statement

Problem Statement:

Twitter, a popular microblogging and social networking platform, generates a massive volume of tweets every second. However, many tweets do not contain hashtags or may use inaccurate or misspelled ones, making it challenging to categorize and analyze the content effectively. To address this issue, we aim to develop Named Entity Recognition (NER) models specifically tailored for Twitter data.

The objective is to automatically identify and tag named entities within tweets, including but not limited to persons, geo-locations, companies, facilities, products, music artists, movies, sports teams, and TV shows. By accurately recognizing named entities, we can gain insights into the trends and topics discussed on Twitter, independent of hashtags.

To achieve this goal, we will follow a systematic approach:

1. Import and explore the provided dataset, which is annotated with 10 fine-grained NER categories, in the CoNLL format.
2. Preprocess the data to prepare it for model training, ensuring it is in the required format.
3. Train a Long Short-Term Memory (LSTM) model with a Conditional Random Field (CRF) layer, utilizing word2vec embeddings for initialization.
4. Train a TensorFlow tokenizer to tokenize the data for training.
5. Split the data into training and testing sets.
6. Train the LSTM + CRF model on the training data, experimenting with various hyperparameters to optimize performance.
7. Evaluate the model's effectiveness by aligning predicted labels with input tokens and computing relevant metrics.
8. Load a pre-trained Transformer model, such as 'bert-base-uncased', from the Transformers library.
9. Obtain the tokenizer for the Transformer model and tokenize the data accordingly.
10. Train the Transformer model on the tokenized data, experimenting with hyperparameters to improve results.
11. Align sub-token outputs to obtain final predictions for named entities.
12. Validate the trained models by making predictions on sample sentences and assessing their performance.

By developing and training NER models tailored for Twitter data, we aim to enhance our understanding of the content shared on the platform, enabling better trend analysis and topic identification.

1. Import & Explore the dataset

```
# Import necessary libraries
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# Define file paths
train_file_path = '/content/drive/MyDrive/Twitter/wnut16.txt.conll'
test_file_path = '/content/drive/MyDrive/Twitter/wnut16test.txt.conll'

# Load the training dataset
with open(train_file_path, 'r') as train_file:
    train_data = train_file.readlines()

# Load the testing dataset
with open(test_file_path, 'r') as test_file:
    test_data = test_file.readlines()
```

```
# Display the first few lines of the training dataset
print("First few lines of the training dataset:")
for line in train_data[:5]:
    print(line.strip())

# Display the first few lines of the testing dataset
print("\nFirst few lines of the testing dataset:")
for line in test_data[:5]:
    print(line.strip())
```

```
First few lines of the training dataset:
@SammieLynnsMom O
@tg10781        O
they    O
will    O
be      O

First few lines of the testing dataset:
New     B-other
Orleans I-other
Mother  I-other
's      I-other
Day     I-other
```

```
# Convert lists to DataFrames
train_df = pd.DataFrame(train_data)
test_df = pd.DataFrame(test_data)

# Check the shape of train and test datasets
print("\nTrain Dataset Shape:", train_df.shape)
print("Test Dataset Shape:", test_df.shape)
```

```
Train Dataset Shape: (48862, 1)
Test Dataset Shape: (65757, 1)
```

```
# Check for missing values
print("Missing values in Train Dataset:")
print(train_df.isnull().sum())

print("\nMissing values in Test Dataset:")
print(test_df.isnull().sum())
```

```
Missing values in Train Dataset:
0    0
dtype: int64

Missing values in Test Dataset:
0    0
dtype: int64
```

2. Data Preprocessing

a. Tokenize the tweets: Split each tweet into individual words or tokens.

b. Encode the tokens and labels: Convert the tokens and corresponding labels into numerical representations that can be fed into the models

```
# Define a tokenizer
tokenizer = tf.keras.preprocessing.text.Tokenizer(oov_token='<UNK>')

# Fit tokenizer on the training data
tokenizer.fit_on_texts(train_data)

# Tokenize and encode the training data
train_sequences = tokenizer.texts_to_sequences(train_data)

# Tokenize and encode the testing data
test_sequences = tokenizer.texts_to_sequences(test_data)
```

```
# Display tokenized sequences for a sample tweet from the training data
sample_tweet_idx = 0
print("Sample tweet before tokenization:", train_data[sample_tweet_idx])

print("Tokenized sequence:", train_sequences[sample_tweet_idx])
```

```
Sample tweet before tokenization: @SammieLynnsMom       O

Tokenized sequence: [2961, 2]
```

```
# Display tokenized sequences for a sample tweet from the testing data
print("Sample tweet before tokenization:", test_data[sample_tweet_idx])
print("Tokenized sequence:", test_sequences[sample_tweet_idx])
```

```
Sample tweet before tokenization: New   B-other

Tokenized sequence: [105, 4, 9]
```

3. Training an LSTM + CRF model

- To initialize embeddings for our LSTM model, we'll use a pre-trained word embedding model such as `Word2Vec`.
- `Word2Vec` embeddings capture semantic relationships between words based on their context in large text corpora, making them suitable for our NER task.

- Install `gensim` library

```
!pip install gensim
```

```
# Import library
import gensim.downloader as api

# Download and load the pre-trained Word2Vec model
word2vec_model = api.load('word2vec-google-news-300')
```

- Next, we'll map the tokens in our dataset to their corresponding embeddings using the loaded Word2Vec model.
- We'll create an embedding matrix where each row corresponds to the embedding vector for a token.

```
# Initialize an empty embedding matrix
embedding_matrix = np.zeros((len(tokenizer.word_index) + 1, 300))  # Assuming 300-dimensional embeddings

# Map tokens to their corresponding embeddings
for word, i in tokenizer.word_index.items():
    if word in word2vec_model:
        embedding_matrix[i] = word2vec_model[word]
```

```
!pip install tensorflow_addons
```

```
# Import necessary libraries
from tensorflow.keras.layers import Input, Embedding, Bidirectional, LSTM, TimeDistributed, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import tensorflow_addons as tfa
```

```
/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(
```

```
# Define model architecture
def build_lstm_crf_model(embedding_matrix, num_classes):
    input_layer = Input(shape=(None,))
    embedding_layer = Embedding(
        input_dim=embedding_matrix.shape[0],
        output_dim=embedding_matrix.shape[1],
        weights=[embedding_matrix],
        trainable=False
    )(input_layer)
    lstm_layer = Bidirectional(LSTM(units=50, return_sequences=True))(embedding_layer)
    crf_layer = tfa.layers.CRF(num_classes)(lstm_layer)
    model = Model(inputs=input_layer, outputs=crf_layer)
    return model


# Define model parameters
num_classes = 11  # Number of NER classes (including 'O' for no entity)
embedding_matrix = embedding_matrix  # Initialized embedding matrix


# Build and compile the model
lstm_crf_model = build_lstm_crf_model(embedding_matrix, num_classes)
lstm_crf_model.compile(optimizer=Adam(learning_rate=0.001), loss='sparse_categorical_crossentropy')


# Display model summary
print("LSTM + CRF Model Summary:\n")
print(lstm_crf_model.summary())
```

```
LSTM + CRF Model Summary:

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, None)]            0         
                                                                 
 embedding (Embedding)       (None, None, 300)         2591700   
                                                                 
 bidirectional (Bidirection  (None, None, 100)         140400    
 al)                                                             
                                                                 
 crf (CRF)                   [(None, None),            1254      
                              (None, None, 11),                  
                              (None,),                           
                              (11, 11)]                          
                                                                 
=================================================================
Total params: 2733354 (10.43 MB)
Trainable params: 141654 (553.34 KB)
Non-trainable params: 2591700 (9.89 MB)
_________________________________________________________________
None
```

In this code:

1. **Importing Libraries**:
   - TensorFlow and its components: `tensorflow`, `tensorflow.keras.layers`, `tensorflow.keras.models`, `tensorflow.keras.optimizers`.
   - TensorFlow Addons for the CRF layer: `tensorflow_addons`.
2. **Custom CRF Layer** (`CRFLayer`):
   - A custom Keras layer is defined (`CRFLayer`) to incorporate the CRF functionality into the neural network architecture.
   - The `CRFLayer` class inherits from `tf.keras.layers.Layer`.
   - Within the `__init__` method, an instance of the CRF layer from TensorFlow Addons (`tfa.layers.CRF`) is created.
   - The `call` method is overridden to define the layer's forward pass. It passes the inputs through the CRF layer.
3. **Model Architecture** (`build_lstm_crf_model`):
   - This function defines the architecture of the LSTM-CRF model.
   - It takes the embedding matrix and the number of classes as input parameters.
   - The input layer is defined with `Input` from `tf.keras.layers`.
   - An embedding layer is added to map the input tokens to dense vectors.
   - Bidirectional LSTM layer processes the embedded sequences.
   - The custom CRF layer (`CRFLayer`) is added to the model to perform sequence labeling with CRF.
4. **Model Compilation**:
   - The model is compiled using the Adam optimizer (`Adam`) with a specified learning rate.
   - The loss function is set to `'sparse_categorical_crossentropy'`.
5. **Model Summary**:

- The summary of the LSTM-CRF model is printed, showing the architecture and the number of parameters.

Overall, the code sets up a neural network architecture for sequence labeling, combining an LSTM layer with a CRF layer for better handling of sequential data. It leverages TensorFlow and TensorFlow Addons to build and train the model efficiently.

4. Train a Tensorflow Tokenizer

- To train the model, we'll use the tokenized and encoded training data `(train_sequences)`. We'll also use the corresponding labels for training.

```
# Import necessary libraries
import re
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.losses import BinaryCrossentropy


# Function to clean words
def clean_word(word):
    return re.sub(r'[^a-zA-Z0-9]', '', word)


train_sequences = []
train_labels = []
current_sequence = []
current_labels = []

for line in train_data:
    if line.strip() == '':  # Check for empty line indicating end of sequence
        if current_sequence and current_labels:  # Check if both sequence and labels are non-empty
            train_sequences.append(current_sequence)
            train_labels.append(current_labels)
        current_sequence = []
        current_labels = []
    else:
        parts = line.strip().split()
        if len(parts) == 2:  # Check if line contains valid token-label pair
            word, label = parts
            word = clean_word(word)
            if word:  # Check if word is not empty after cleaning
                current_sequence.append(word)
                current_labels.append(label)

# Ensure the last sequence is added if file does not end with a newline
if current_sequence and current_labels:
    train_sequences.append(current_sequence)
    train_labels.append(current_labels)


# Map labels to integers
label_map = {label: idx for idx, label in enumerate(set(sum(train_labels, [])))}
y_train = [[label_map[label] for label in sequence] for sequence in train_labels]

# Define the maximum sequence length
max_sequence_length = 100

# Convert tokenized sequences to numpy arrays and pad sequences
X_train = pad_sequences([[int(hash(word)) for word in seq] for seq in train_sequences], padding='post')
y_train = pad_sequences(y_train, padding='post', maxlen=max_sequence_length)

# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=len(label_map))

# Ensure X_train and y_train have the correct shapes
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
```

```
X_train shape: (2394, 36)
y_train shape: (2394, 100, 21)
```

```
# Define lstm_crf_model, batch_size, and epochs
# These are placeholders and should be replaced with your actual model and parameters
batch_size = 32  # Example batch size
epochs = 10     # Example number of epochs

# Define vocabulary size, embedding dimension, and maximum sequence length
vocab_size = 1000000
embedding_dim = 128
max_sequence_length = X_train.shape[1]

# Rebuild the model with the updated vocabulary size
input_layer = tf.keras.layers.Input(shape=(max_sequence_length,))

# Embedding layer
embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_dim)(input_layer)

# LSTM layer
lstm_layer = LSTM(units=64, return_sequences=True)(embedding_layer)

# Dense layer
dense_layer = Dense(len(label_map), activation='relu')(lstm_layer)

# CRF layer
crf_layer = tfa.layers.CRF(len(label_map))(dense_layer)

# Create the model
lstm_crf_model = tf.keras.models.Model(inputs=input_layer, outputs=crf_layer)
```

```
# Define the loss function with the corrected epsilon casting
epsilon = tf.constant(0, dtype=tf.int32)

# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=len(label_map))

# Define loss function
loss_function = tfa.losses.SigmoidFocalCrossEntropy(from_logits=True)

# Using the crf log likelihood as loss function
def crf_loss(y_true, y_pred):
    # Add a batch dimension to y_true if it doesn't have one
    if len(y_true.shape) == 2:
        y_true = tf.expand_dims(y_true, axis=-1)

    # Return the negative log-likelihood of the CRF layer
    return -y_pred[1]

# Compile the model
lstm_crf_model.compile(optimizer='adam', loss=crf_loss, metrics=[tfa.metrics.CohenKappa(num_classes=len(label_map))])

# Print model summary
lstm_crf_model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 36)]              0

 embedding_1 (Embedding)     (None, 36, 128)           128000000

 lstm_1 (LSTM)               (None, 36, 64)            49408

 dense_1 (Dense)             (None, 36, 21)            1365

 crf_1 (CRF)                 [(None, 36),              945
                              (None, 36, 21),
                              (None,),
                              (21, 21)]

=================================================================
Total params: 128051718 (488.48 MB)
Trainable params: 128051718 (488.48 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In this code:

Sure! Let's go through the code step by step to understand its functionality and purpose.

1. Import Necessary Libraries

- This section imports the necessary libraries for the task. `re` is used for regular expressions, `tensorflow.keras` modules for building and training the model, and `tensorflow_addons` for the CRF layer.

2. Function to Clean Words

- This function removes any special characters from a given word, leaving only alphanumeric characters.

3. Parsing Tokenized Sequences and Labels

- This code parses the `train_data`, which is expected to be a list of strings representing lines from a text file.
- Each line contains a word and its corresponding label. The sequences are separated by empty lines.
- The cleaned words and their labels are stored in `train_sequences` and `train_labels`, respectively.

4. Mapping Labels to Integers

- This code creates a mapping from labels to integer indices (`label_map`). It then converts the list of label sequences (`train_labels`) into sequences of integers (`y_train`) using this mapping.

5. Padding Sequences

- This section converts the tokenized sequences into numpy arrays and pads them to ensure they all have the same length (`max_sequence_length`).

6. One-Hot Encoding Labels

- This converts the integer labels into one-hot encoded vectors.

7. Model Definition

- This section defines the architecture of the LSTM-CRF model:
  - `input_layer` specifies the shape of the input data.
  - `embedding_layer` maps input words to dense vectors of fixed size.
  - `lstm_layer` processes the sequences using an LSTM network.
  - `dense_layer` applies a dense (fully connected) layer to the output of the LSTM.
  - `crf_layer` applies a CRF layer to model sequential dependencies.

8. Model Compilation

- This section compiles the model is compiled with a loss function with CRF layers and multiclass classification. The model's summary is printed to display its architecture.

5. Train-Test Split the Data

```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train_split, X_test_split, y_train_split, y_test_split = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

print(f"Train Data Shape: {X_train_split.shape}")
print(f"Test Data Shape: {X_test_split.shape}")
```

```
Train Data Shape: (1915, 36)
Test Data Shape: (479, 36)
```