Data Preprocessing

```
import os

def load_conll_data(file_path):
    """
    Loads data from a CoNLL formatted file.

    Args:
        file_path (str): The path to the CoNLL file.

    Returns:
        tuple: A tuple containing two lists:
               - sentences (list of lists of words)
               - tags (list of lists of tags)
    """
    sentences = []
    tags = []
    current_sentence = []
    current_tags = []

    if not os.path.exists(file_path):
        print(f"Error: File not found at {file_path}")
        return [], []

    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                line = line.strip()
                if line == "": # End of a sentence
                    if current_sentence:
                        sentences.append(current_sentence)
                        tags.append(current_tags)
                        current_sentence = []
                        current_tags = []
                else:
                    parts = line.split() # Default split handles potential tabs/spaces
                    if len(parts) >= 2:
                        word = parts[0]
                        tag = parts[-1] # Assume tag is the last element
                        current_sentence.append(word)
                        current_tags.append(tag)
                    else:
                        # Handle potential malformed lines, e.g., lines with only a word or tag
                        print(f"Skipping malformed line: '{line}' in file {file_path}")

            # Add the last sentence if the file doesn't end with a blank line
            if current_sentence:
                sentences.append(current_sentence)
                tags.append(current_tags)

    except Exception as e:
        print(f"Error reading file {file_path}: {e}")
        return [], []

    return sentences, tags

def get_data_stats(sentences, tags, dataset_name="Dataset"):
    """Calculates and prints basic statistics about the loaded data."""
    num_sentences = len(sentences)
    num_tokens = sum(len(s) for s in sentences)
    all_tags = [tag for tag_list in tags for tag in tag_list]
    unique_tags = sorted(list(set(all_tags)))
    num_unique_tags = len(unique_tags)

    print(f"--- {dataset_name} Statistics ---")
    print(f"Number of sentences: {num_sentences}")
    print(f"Number of tokens: {num_tokens}")
    print(f"Number of unique tags: {num_unique_tags}")
    print(f"Unique tags: {unique_tags}")
    print("-" * (len(dataset_name) + 18))
    return unique_tags

if __name__ == "__main__":
    train_file = "wnut 16.txt.conll"
    test_file = "wnut 16test.txt.conll"

    print(f"Loading training data from: {train_file}")
    train_sentences, train_tags = load_conll_data(train_file)
```

```
    print(f"\nLoading test data from: {test_file}")
    test_sentences, test_tags = load_conll_data(test_file)

    if train_sentences and test_sentences:
        print("\nCalculating statistics...")
        train_unique_tags = get_data_stats(train_sentences, train_tags, "Training Set")
        test_unique_tags = get_data_stats(test_sentences, test_tags, "Test Set")

        # Check if tag sets are consistent (optional but good practice)
        if set(train_unique_tags) == set(test_unique_tags):
            print("\nTag sets are consistent between training and test data.")
        else:
            print("\nWarning: Tag sets differ between training and test data.")
            print(f"Tags only in train: {set(train_unique_tags) - set(test_unique_tags)}")
            print(f"Tags only in test: {set(test_unique_tags) - set(train_unique_tags)}")

        print("\nSample Data (First sentence):")
        if train_sentences:
            print("Train Sentence:", train_sentences[0])
            print("Train Tags:", train_tags[0])
        if test_sentences:
            print("Test Sentence:", test_sentences[0])
            print("Test Tags:", test_tags[0])

    else:
        print("\nCould not load data properly. Please check file paths and format.")
```

```
Loading training data from: wnut 16.txt.conll

Loading test data from: wnut 16test.txt.conll

Calculating statistics...
--- Training Set Statistics ---
Number of sentences: 2394
Number of tokens: 46469
Number of unique tags: 21
Unique tags: ['B-company', 'B-facility', 'B-geo-loc', 'B-movie', 'B-musicartist', 'B-other', 'B-person', 'B-product', 'B-sportsteam
------------------------------
--- Test Set Statistics ---
Number of sentences: 3850
Number of tokens: 61908
Number of unique tags: 21
Unique tags: ['B-company', 'B-facility', 'B-geo-loc', 'B-movie', 'B-musicartist', 'B-other', 'B-person', 'B-product', 'B-sportsteam
-------------------------

Tag sets are consistent between training and test data.

Sample Data (First sentence):
Train Sentence: ['@SammieLynnsMom', '@tg10781', 'they', 'will', 'be', 'all', 'done', 'by', 'Sunday', 'trust', 'me', '*wink*']
Train Tags: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Test Sentence: ['New', 'Orleans', 'Mother', "'s", 'Day', 'Parade', 'shooting', '.', 'One', 'of', 'the', 'people', 'hurt', 'was', 'a
Test Tags: ['B-other', 'I-other', 'I-other', 'I-other', 'I-other', 'I-other', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
```

Train BERT NER

```
import os
import pickle
import numpy as np
from transformers import BertTokenizerFast
from torch.utils.data import Dataset, DataLoader
import torch

# --- Data Loading Function (copied from preprocess_data.py) ---
def load_conll_data(file_path):
    """Loads data from a CoNLL formatted file."""
    sentences = []
    tags = []
    current_sentence = []
    current_tags = []
    if not os.path.exists(file_path):
        print(f"Error: File not found at {file_path}")
        return [], []
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                line = line.strip()
                if line == "":
                    if current_sentence:
                        sentences.append(current_sentence)
```

```
                        tags.append(current_tags)
                        current_sentence = []
                        current_tags = []
                else:
                    parts = line.split()
                    if len(parts) >= 2:
                        word = parts[0]
                        tag = parts[-1]
                        current_sentence.append(word)
                        current_tags.append(tag)
                    else:
                        print(f"Skipping malformed line: '{line}' in file {file_path}")
            if current_sentence:
                sentences.append(current_sentence)
                tags.append(current_tags)
    except Exception as e:
        print(f"Error reading file {file_path}: {e}")
        return [], []
    return sentences, tags
# --- End of Data Loading Function ---

# --- Configuration ---
TRAIN_FILE = "wnut 16.txt.conll"
TEST_FILE = "wnut 16test.txt.conll"
MODEL_NAME = 'bert-base-uncased'
MAX_LEN = 128 # Max sequence length for BERT
BATCH_SIZE = 16 # Adjust based on GPU memory
TAG_MAP_PATH = "tag_map_bert.pkl" # Separate tag map for BERT potentially
PREPARED_BERT_DATA_PATH = "prepared_bert_data.pkl"

# --- 1. Load Data ---
print("Loading data...")
train_sentences, train_tags = load_conll_data(TRAIN_FILE)
test_sentences, test_tags = load_conll_data(TEST_FILE) # Using test set for now, will split train later

if not train_sentences or not test_sentences:
    print("Failed to load data. Exiting.")
    exit()

print(f"Loaded {len(train_sentences)} training sentences and {len(test_sentences)} test sentences.")

# --- 2. Create Tag Mapping ---
print("\nCreating tag mapping...")
# Use tags from training data only to define the mapping
all_tags_flat_train = [tag for sublist in train_tags for tag in sublist]
unique_tags = sorted(list(set(all_tags_flat_train)))
tag2idx = {tag: i for i, tag in enumerate(unique_tags)}
idx2tag = {i: tag for tag, i in tag2idx.items()}
n_tags = len(unique_tags)
print(f"Number of unique tags (from train set): {n_tags}")
print(f"Tag mapping: {tag2idx}")

# Save tag mapping
with open(TAG_MAP_PATH, 'wb') as f:
    pickle.dump({'tag2idx': tag2idx, 'idx2tag': idx2tag, 'n_tags': n_tags}, f)
print(f"Tag mapping saved to {TAG_MAP_PATH}")

# --- 3. Tokenization and Label Alignment ---
print(f"\nLoading tokenizer: {MODEL_NAME}...")
tokenizer = BertTokenizerFast.from_pretrained(MODEL_NAME)

def tokenize_and_align_labels(sentences, tags, tokenizer, tag2idx_map):
    """
    Tokenizes sentences using the provided tokenizer and aligns the
    corresponding NER tags to the generated tokens (WordPieces/subwords).
    Uses -100 for special tokens and subsequent subword tokens,
    so they are ignored by the loss function during training.
    """
    # Ensure 'O' tag exists for default assignment
    o_tag_idx = tag2idx_map.get('O')
    if o_tag_idx is None:
        # This should ideally not happen if 'O' is in the training data,
        # but handle it just in case. Assign a default index (e.g., 0)
        # or raise an error. Here, we'll print a warning and use 0.
        print("Warning: 'O' tag not found in tag2idx mapping. Using index 0 as default.")
        o_tag_idx = 0 # Or choose another appropriate default/error handling

    tokenized_inputs = tokenizer(sentences, truncation=True, is_split_into_words=True, padding='max_length', max_length=MAX_LEN)
    labels = []
    for i, label_list in enumerate(tags):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx = None
```

```
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None: # Special tokens like [CLS], [SEP]
                label_ids.append(-100) # Ignore special tokens in loss calculation
            elif word_idx != previous_word_idx: # First token of a new word
                # Assign the label of the current word
                # Use .get() with default 'O' tag index for robustness
                label_ids.append(tag2idx_map.get(label_list[word_idx], o_tag_idx))
            else: # Subsequent tokens of the same word
                # Assign -100 to ignore these subword tokens in loss calculation
                label_ids.append(-100)
            previous_word_idx = word_idx
        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs

print("Tokenizing and aligning labels for training set...")
train_encodings = tokenize_and_align_labels(train_sentences, train_tags, tokenizer, tag2idx)
print("Tokenizing and aligning labels for test set...")
test_encodings = tokenize_and_align_labels(test_sentences, test_tags, tokenizer, tag2idx)

# --- 4. Create PyTorch Dataset ---
class NERDataset(Dataset):
    def __init__(self, encodings):
        self.encodings = encodings

    def __getitem__(self, idx):
        # Return tensors
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        return item

    def __len__(self):
        return len(self.encodings.labels)

train_dataset = NERDataset(train_encodings)
test_dataset = NERDataset(test_encodings) # Will split train_dataset later for validation

print(f"\nSample Train Encoding (Input IDs): {train_dataset[0]['input_ids']}")
print(f"Sample Train Encoding (Labels): {train_dataset[0]['labels']}")

# --- 5. Save Processed Data ---
# Save datasets and tokenizer info for the training script
processed_data = {
    'train_dataset': train_dataset,
    'test_dataset': test_dataset, # Note: This is the original test set
    'tag2idx': tag2idx,
    'idx2tag': idx2tag,
    'n_tags': n_tags
}
with open(PREPARED_BERT_DATA_PATH, 'wb') as f:
    pickle.dump(processed_data, f)
print(f"\nProcessed data saved to {PREPARED_BERT_DATA_PATH}")

print("\nBERT data preparation finished.")
```

```
Loading data...
Loaded 2394 training sentences and 3850 test sentences.

Creating tag mapping...
Number of unique tags (from train set): 21
Tag mapping: {'B-company': 0, 'B-facility': 1, 'B-geo-loc': 2, 'B-movie': 3, 'B-musicartist': 4, 'B-other': 5, 'B-person': 6, 'B-pro
Tag mapping saved to tag_map_bert.pkl

Loading tokenizer: bert-base-uncased...
Tokenizing and aligning labels for training set...
Tokenizing and aligning labels for test set...

Sample Train Encoding (Input IDs): tensor([  101,  1030,  3520,  9856, 27610, 25855,  2213,  1030,  1056,  2290,
        10790,  2581,  2620,  2487,  2027,  2097,  2022,  2035,  2589,  2011,
         4465,  3404,  2033,  1008, 16837,  1008,   102,     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,     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])
Sample Train Encoding (Labels): tensor([-100,   20, -100, -100, -100, -100, -100,   20, -100, -100, -100, -100,
        -100, -100,   20,   20,   20,   20,   20,   20,   20,   20,   20,   20,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
```

```
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
            -100, -100, -100, -100, -100, -100, -100, -100])

    Processed data saved to prepared_bert_data.pkl

    BERT data preparation finished.
```

```
pip install seqeval
```

```
Requirement already satisfied: seqeval in /usr/local/lib/python3.11/dist-packages (1.2.2)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.11/dist-packages (from seqeval) (1.26.4)
Requirement already satisfied: scikit-learn>=0.21.3 in /usr/local/lib/python3.11/dist-packages (from seqeval) (1.6.1)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.21.3->seqeval) (1.13.1
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.21.3->seqeval) (1.4.2
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.21.3->seqeval)
```

Run BERT Training

```
import pickle
import torch
from torch.utils.data import DataLoader, random_split, Dataset # Added Dataset import back
from transformers import BertForTokenClassification, TrainingArguments, Trainer, BertTokenizerFast # Removed AdamW, Added BertTokenizerF
import numpy as np
from seqeval.metrics import classification_report, f1_score, accuracy_score

# --- Configuration ---
PREPARED_BERT_DATA_PATH = "prepared_bert_data.pkl"
MODEL_NAME = 'bert-base-uncased'
OUTPUT_DIR = './bert_ner_output' # Directory to save model checkpoints and results
LOGGING_DIR = './bert_ner_logs' # Directory for TensorBoard logs
MODEL_SAVE_PATH = './bert_ner_final_model' # Directory to save the final fine-tuned model

# Training Hyperparameters (can be tuned)
LEARNING_RATE = 3e-5
EPOCHS = 3 # Fewer epochs usually needed for fine-tuning BERT
BATCH_SIZE = 16 # Adjust based on GPU memory
WEIGHT_DECAY = 0.01
VALIDATION_SPLIT_RATIO = 0.1 # Use 10% of training data for validation

# --- Re-define NERDataset class (needed if loading from pickle) ---
class NERDataset(Dataset):
    def __init__(self, encodings):
        self.encodings = encodings

    def __getitem__(self, idx):
        # Return tensors
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        return item

    def __len__(self):
        return len(self.encodings.labels)

# --- 1. Load Processed Data ---
print("Loading processed data...")
with open(PREPARED_BERT_DATA_PATH, 'rb') as f:
    processed_data = pickle.load(f)

# The datasets are already NERDataset objects from the previous script
train_dataset_full = processed_data['train_dataset']
test_dataset = processed_data['test_dataset']
tag2idx = processed_data['tag2idx']
idx2tag = processed_data['idx2tag']
n_tags = processed_data['n_tags']

print("Data loaded successfully.")

# --- 2. Split Training Data into Train/Validation ---
print("Splitting training data into train/validation sets...")
train_size = int((1.0 - VALIDATION_SPLIT_RATIO) * len(train_dataset_full))
val_size = len(train_dataset_full) - train_size
train_dataset, val_dataset = random_split(train_dataset_full, [train_size, val_size])
print(f"Train size: {len(train_dataset)}, Validation size: {len(val_dataset)}, Test size: {len(test_dataset)}")

# --- 3. Load Pre-trained Model ---
```

```
print(f"\nLoading pre-trained model: {MODEL_NAME}...")
model = BertForTokenClassification.from_pretrained(MODEL_NAME, num_labels=n_tags)

# --- 4. Define Metrics Computation ---
def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    # Remove ignored index (-100) and convert indices to labels
    true_labels = []
    true_predictions = []
    for prediction_list, label_list in zip(predictions, labels):
        temp_true = []
        temp_pred = []
        for prediction, label in zip(prediction_list, label_list):
            if label != -100: # Only consider non-ignored labels
                temp_true.append(idx2tag[label])
                temp_pred.append(idx2tag[prediction])
        true_labels.append(temp_true)
        true_predictions.append(temp_pred)

    # Use seqeval to compute metrics
    report = classification_report(true_labels, true_predictions, output_dict=True, zero_division=0)

    # Extract overall metrics (micro avg is common for NER)
    results = {
        "precision": report["micro avg"]["precision"],
        "recall": report["micro avg"]["recall"],
        "f1": report["micro avg"]["f1-score"],
        "accuracy": accuracy_score(true_labels, true_predictions),
    }
    return results

# --- 5. Define Training Arguments ---
print("\nSetting up training arguments...")
training_args = TrainingArguments(
    output_dir=OUTPUT_DIR,
    num_train_epochs=EPOCHS,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    learning_rate=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY,
    logging_dir=LOGGING_DIR,
    logging_steps=50, # Log metrics less frequently
    evaluation_strategy="epoch", # Evaluate at the end of each epoch
    save_strategy="epoch", # Save model checkpoint at the end of each epoch
    load_best_model_at_end=True, # Load the best model based on validation loss at the end
    metric_for_best_model="eval_loss", # Use validation loss to determine the best model
    greater_is_better=False, # Lower validation loss is better
    report_to="tensorboard" # Log to TensorBoard
)

# --- 6. Initialize Trainer ---
print("Initializing Trainer...")
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
)

# --- 7. Train the Model ---
print("\nStarting training...")
trainer.train()
print("Training finished.")

# --- 8. Evaluate on Test Set ---
print("\nEvaluating on test set...")
test_results = trainer.evaluate(eval_dataset=test_dataset)
print("\nTest Set Evaluation Results:")
print(test_results)

# --- 9. Save Final Model and Tokenizer ---
print(f"\nSaving final model to {MODEL_SAVE_PATH}...")
trainer.save_model(MODEL_SAVE_PATH)
# Tokenizer was loaded in the previous script, saving it here too for completeness
tokenizer = BertTokenizerFast.from_pretrained(MODEL_NAME) # Re-load tokenizer
tokenizer.save_pretrained(MODEL_SAVE_PATH)
print("Model and tokenizer saved successfully.")
```

```
print("\nBERT fine-tuning and evaluation complete.")
```

```
Loading processed data...
Data loaded successfully.
Splitting training data into train/validation sets...
Train size: 2154, Validation size: 240, Test size: 3850

Loading pre-trained model: bert-base-uncased...
Some weights of BertForTokenClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initial
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.11/dist-packages/transformers/training_args.py:1611: FutureWarning: `evaluation_strategy` is deprecated and v
  warnings.warn(

Setting up training arguments...
Initializing Trainer...

Starting training...
[405/405 02:40, Epoch 3/3]
```

Epoch	Training Loss	Validation Loss	Precision	Recall	F1	Accuracy
1	0.260100	0.256610	0.278195	0.217647	0.244224	0.946957
2	0.155200	0.208706	0.553957	0.452941	0.498382	0.958580
3	0.108700	0.213389	0.491525	0.511765	0.501441	0.957523

```
Training finished.

Evaluating on test set...
[241/241 00:26]

Test Set Evaluation Results:
{'eval_loss': 0.34703338146209717, 'eval_precision': 0.30686695278969955, 'eval_recall': 0.2470486610999136, 'eval_f1': 0.273727867

Saving final model to ./bert_ner_final_model...
Model and tokenizer saved successfully.

BERT fine-tuning and evaluation complete.
```

Predict BERT

```
import torch
from transformers import BertTokenizerFast, BertForTokenClassification
import pickle
import numpy as np

# --- Configuration ---
MODEL_PATH = './bert_ner_final_model' # Path where the fine-tuned model and tokenizer are saved
TAG_MAP_PATH = "tag_map_bert.pkl"

# --- Load Model, Tokenizer, and Tag Mapping ---
print(f"Loading model and tokenizer from {MODEL_PATH}...")
model = BertForTokenClassification.from_pretrained(MODEL_PATH)
tokenizer = BertTokenizerFast.from_pretrained(MODEL_PATH)

print(f"Loading tag mapping from {TAG_MAP_PATH}...")
with open(TAG_MAP_PATH, 'rb') as f:
    tag_maps = pickle.load(f)
idx2tag = tag_maps['idx2tag']
tag2idx = tag_maps['tag2idx'] # Needed for potential checks, though idx2tag is primary for output

# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(f"Using device: {device}")

# --- Prediction Function ---
def predict_ner(sentence, model, tokenizer, idx2tag_map):
    """Predicts NER tags for a given sentence."""
    # Tokenize the sentence
    inputs = tokenizer(sentence, return_tensors="pt", truncation=True, padding=True)

    # Move inputs to the correct device
    input_ids = inputs["input_ids"].to(device)
    attention_mask = inputs["attention_mask"].to(device)

    # Get model predictions
    model.eval() # Set model to evaluation mode
    with torch.no_grad():
        outputs = model(input_ids, attention_mask=attention_mask)
```

```
        logits = outputs.logits

    # Get the most likely tag index for each token
    predictions = torch.argmax(logits, dim=2)

    # Convert token IDs and prediction indices back to words and tags
    tokens = tokenizer.convert_ids_to_tokens(input_ids[0].cpu().numpy())
    predicted_indices = predictions[0].cpu().numpy()

    # Align tokens and predictions (ignoring special tokens and padding)
    word_tags = []
    current_word = ""
    current_tag_idx = -1 # Initialize with an invalid index

    # Use word_ids to group subword tokens
    word_ids = inputs.word_ids(batch_index=0)
    previous_word_idx = None

    for i, token in enumerate(tokens):
        if token in [tokenizer.cls_token, tokenizer.sep_token, tokenizer.pad_token]:
            continue

        word_idx = word_ids[i]
        tag_idx = predicted_indices[i]

        if word_idx is None: # Should not happen if we skip special tokens, but good check
             continue

        # If it's the start of a new word (or the first word)
        if word_idx != previous_word_idx:
            # Add the previous word and its tag if it exists
            if current_word:
                 word_tags.append((current_word, idx2tag_map.get(current_tag_idx, 'O'))) # Default to 'O'

            # Start the new word
            current_word = token
            current_tag_idx = tag_idx
        else: # It's a subword token, append to the current word
            # Remove '##' prefix if present
            current_word += token.replace('##', '')
            # Keep the tag of the first subword token (common strategy)
            # current_tag_idx remains unchanged

        previous_word_idx = word_idx

    # Add the last word
    if current_word:
         word_tags.append((current_word, idx2tag_map.get(current_tag_idx, 'O')))

    return word_tags

# --- Example Sentences ---
sentences_to_predict = [
    "Harry Potter went to London to watch the Arsenal game.",
    "Apple announced the new iPhone at the Steve Jobs Theater in Cupertino.",
    "Taylor Swift released her album 'Folklore' last year.",
    "Watching The Office on Netflix is my favorite pastime."
]

# --- Perform Predictions ---
print("\nPerforming predictions on custom sentences:")
for sentence in sentences_to_predict:
    print(f"\nSentence: {sentence}")
    predicted_tags = predict_ner(sentence, model, tokenizer, idx2tag)
    print("Predicted Tags:", predicted_tags)

print("\nPrediction complete.")
```

```
Loading model and tokenizer from ./bert_ner_final_model...
Loading tag mapping from tag_map_bert.pkl...
Using device: cuda

Performing predictions on custom sentences:

Sentence: Harry Potter went to London to watch the Arsenal game.
Predicted Tags: [('harry', 'B-person'), ('potter', 'I-person'), ('went', 'O'), ('to', 'O'), ('london', 'B-geo-loc'), ('to', 'O'), (

Sentence: Apple announced the new iPhone at the Steve Jobs Theater in Cupertino.
Predicted Tags: [('apple', 'B-company'), ('announced', 'O'), ('the', 'O'), ('new', 'O'), ('iphone', 'O'), ('at', 'O'), ('the', 'O'),

Sentence: Taylor Swift released her album 'Folklore' last year.
Predicted Tags: [('taylor', 'B-person'), ('swift', 'I-person'), ('released', 'O'), ('her', 'O'), ('album', 'O'), ("'", 'O'), ('folk]
```

```
Sentence: Watching The Office on Netflix is my favorite pastime.
Predicted Tags: [('watching', 'O'), ('the', 'O'), ('office', 'O'), ('on', 'O'), ('netflix', 'O'), ('is', 'O'), ('my', 'O'), ('favori

Prediction complete.
```

Train LSTM CRF

```
import os
import numpy as np
from gensim.models import Word2Vec
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
import pickle # To save tokenizer and mappings

# --- Data Loading Function (copied from preprocess_data.py) ---
def load_conll_data(file_path):
    """Loads data from a CoNLL formatted file."""
    sentences = []
    tags = []
    current_sentence = []
    current_tags = []
    if not os.path.exists(file_path):
        print(f"Error: File not found at {file_path}")
        return [], []
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                line = line.strip()
                if line == "":
                    if current_sentence:
                        sentences.append(current_sentence)
                        tags.append(current_tags)
                        current_sentence = []
                        current_tags = []
                else:
                    parts = line.split()
                    if len(parts) >= 2:
                        word = parts[0]
                        tag = parts[-1]
                        current_sentence.append(word)
                        current_tags.append(tag)
                    else:
                        print(f"Skipping malformed line: '{line}' in file {file_path}")
            if current_sentence:
                sentences.append(current_sentence)
                tags.append(current_tags)
    except Exception as e:
        print(f"Error reading file {file_path}: {e}")
        return [], []
    return sentences, tags
# --- End of Data Loading Function ---

# --- Configuration ---
TRAIN_FILE = "wnut 16.txt.conll"
TEST_FILE = "wnut 16test.txt.conll"
W2V_MODEL_PATH = "word2vec_lstm.model"
TOKENIZER_PATH = "tokenizer_lstm.pkl"
TAG_MAP_PATH = "tag_map_lstm.pkl"
EMBEDDING_MATRIX_PATH = "embedding_matrix_lstm.npy" # Added path for saving matrix
PREPARED_DATA_PATH = "prepared_lstm_data.pkl" # Added path for saving prepared data

EMBEDDING_DIM = 100  # Dimension for Word2Vec and Embedding layer
MAX_SEQ_LEN = 50     # Maximum sequence length after padding
VALIDATION_SPLIT = 0.2
RANDOM_STATE = 42

# --- 1. Load Data ---
print("Loading data...")
train_sentences, train_tags = load_conll_data(TRAIN_FILE)
test_sentences, test_tags = load_conll_data(TEST_FILE)

if not train_sentences or not test_sentences:
    print("Failed to load data. Exiting.")
    exit()

print(f"Loaded {len(train_sentences)} training sentences and {len(test_sentences)} test sentences.")

# --- 2. Train Word2Vec ---
```

```
print("\nTraining Word2Vec model...")
# Combine train and test sentences for a richer vocabulary
all_sentences = train_sentences + test_sentences
w2v_model = Word2Vec(sentences=all_sentences, vector_size=EMBEDDING_DIM, window=5, min_count=1, workers=4, sg=1) # Using Skip-gram
w2v_model.save(W2V_MODEL_PATH)
print(f"Word2Vec model saved to {W2V_MODEL_PATH}")
print(f"Vocabulary size: {len(w2v_model.wv.index_to_key)}")

# --- 3. Prepare Data for TensorFlow ---
print("\nPreparing data for TensorFlow...")

# 3.1. Create Tag Mapping
all_tags_flat = [tag for sublist in train_tags + test_tags for tag in sublist]
unique_tags = sorted(list(set(all_tags_flat)))
tag2idx = {tag: i for i, tag in enumerate(unique_tags)}
idx2tag = {i: tag for tag, i in tag2idx.items()}
n_tags = len(unique_tags)
print(f"Number of unique tags: {n_tags}")
print(f"Tag mapping: {tag2idx}")

# Save tag mapping
with open(TAG_MAP_PATH, 'wb') as f:
    pickle.dump({'tag2idx': tag2idx, 'idx2tag': idx2tag}, f)
print(f"Tag mapping saved to {TAG_MAP_PATH}")

# 3.2. Tokenize Words
# Use Keras Tokenizer, fit on training sentences only to avoid data leakage
word_tokenizer = Tokenizer(oov_token="<OOV>") # Out-of-vocabulary token
word_tokenizer.fit_on_texts(train_sentences)
vocab_size = len(word_tokenizer.word_index) + 1 # +1 for padding token 0
print(f"Vocabulary size (Keras Tokenizer): {vocab_size}")

# Save tokenizer
with open(TOKENIZER_PATH, 'wb') as f:
    pickle.dump(word_tokenizer, f)
print(f"Tokenizer saved to {TOKENIZER_PATH}")

# 3.3. Convert Sentences and Tags to Sequences
X_train = word_tokenizer.texts_to_sequences(train_sentences)
y_train = [[tag2idx[tag] for tag in tags] for tags in train_tags]

X_test = word_tokenizer.texts_to_sequences(test_sentences)
y_test = [[tag2idx[tag] for tag in tags] for tags in test_tags]

# 3.4. Pad Sequences
print(f"\nPadding sequences to max length: {MAX_SEQ_LEN}...")
X_train_padded = pad_sequences(X_train, maxlen=MAX_SEQ_LEN, padding='post')
y_train_padded = pad_sequences(y_train, maxlen=MAX_SEQ_LEN, padding='post', value=tag2idx['O']) # Pad tags with 'O' tag index

X_test_padded = pad_sequences(X_test, maxlen=MAX_SEQ_LEN, padding='post')
y_test_padded = pad_sequences(y_test, maxlen=MAX_SEQ_LEN, padding='post', value=tag2idx['O'])

print("Padding complete.")

# --- 4. Train/Validation Split ---
print("\nSplitting data into training and validation sets...")
X_train_split, X_val_split, y_train_split, y_val_split = train_test_split(
    X_train_padded, y_train_padded, test_size=VALIDATION_SPLIT, random_state=RANDOM_STATE
)

print(f"Training sequences shape: {X_train_split.shape}")
print(f"Training tags shape: {y_train_split.shape}")
print(f"Validation sequences shape: {X_val_split.shape}")
print(f"Validation tags shape: {y_val_split.shape}")
print(f"Test sequences shape: {X_test_padded.shape}")
print(f"Test tags shape: {y_test_padded.shape}")

# --- 5. Create Embedding Matrix (using Word2Vec) ---
print("\nCreating embedding matrix...")
embedding_matrix = np.zeros((vocab_size, EMBEDDING_DIM))
hits = 0
misses = 0
for word, i in word_tokenizer.word_index.items():
    if word in w2v_model.wv:
        embedding_matrix[i] = w2v_model.wv[word]
        hits += 1
    else:
        misses += 1
        # Words not found in Word2Vec will be initialized to zero vectors.

print(f"Converted {hits} words ({misses} misses)")
print(f"Embedding matrix shape: {embedding_matrix.shape}")
```

```
# Save embedding matrix
np.save(EMBEDDING_MATRIX_PATH, embedding_matrix)
print(f"Embedding matrix saved to {EMBEDDING_MATRIX_PATH}")

# Save prepared data splits for later use in model training script
prepared_data = {
    'X_train': X_train_split,
    'y_train': y_train_split,
    'X_val': X_val_split,
    'y_val': y_val_split,
    'X_test': X_test_padded,
    'y_test': y_test_padded,
    'vocab_size': vocab_size,
    'n_tags': n_tags,
    'max_seq_len': MAX_SEQ_LEN
}
with open(PREPARED_DATA_PATH, 'wb') as f:
    pickle.dump(prepared_data, f)
print(f"Prepared data saved to {PREPARED_DATA_PATH}")


print("\nData preparation finished. Ready for model building and training.")
```

Run LSTM CRF Training

```
import pickle
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Bidirectional, LSTM, TimeDistributed, Dense, Activation
# Note: CRF layer attempts using tensorflow-addons and keras-crf failed due to library incompatibilities/deprecation.
# This script now trains a standard BiLSTM model without a CRF layer.
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from seqeval.metrics import classification_report, f1_score, accuracy_score # Added accuracy_score back for consistency

# --- Configuration ---
PREPARED_DATA_PATH = "prepared_lstm_data.pkl"
EMBEDDING_MATRIX_PATH = "embedding_matrix_lstm.npy"
TAG_MAP_PATH = "tag_map_lstm.pkl"
MODEL_SAVE_PATH = "bilstm_model.h5" # Changed to .h5 format

# Hyperparameters (can be tuned later)
LSTM_UNITS = 64
DROPOUT_RATE = 0.1 # Added dropout for regularization
LEARNING_RATE = 0.001
EPOCHS = 15 # Increased epochs, will use EarlyStopping
BATCH_SIZE = 32
PATIENCE = 3 # For EarlyStopping

# --- 1. Load Prepared Data and Artifacts ---
print("Loading prepared data and artifacts...")
with open(PREPARED_DATA_PATH, 'rb') as f:
    prepared_data = pickle.load(f)

X_train = prepared_data['X_train']
y_train = prepared_data['y_train']
X_val = prepared_data['X_val']
y_val = prepared_data['y_val']
X_test = prepared_data['X_test']
y_test = prepared_data['y_test']
vocab_size = prepared_data['vocab_size']
n_tags = prepared_data['n_tags']
max_seq_len = prepared_data['max_seq_len']

embedding_matrix = np.load(EMBEDDING_MATRIX_PATH)
embedding_dim = embedding_matrix.shape[1] # Get embedding dim from matrix

with open(TAG_MAP_PATH, 'rb') as f:
    tag_maps = pickle.load(f)
idx2tag = tag_maps['idx2tag']

print("Data loaded successfully.")
print(f"Vocab size: {vocab_size}, Embedding dim: {embedding_dim}, Max seq len: {max_seq_len}, Num tags: {n_tags}")

# --- 2. Build BiLSTM Model ---
print("\nBuilding BiLSTM model (CRF attempts failed due to library issues)...")

# Input layer
```

```
input_layer = Input(shape=(max_seq_len,))

# Embedding layer (using pre-trained Word2Vec weights)
# Set trainable=False initially, can be fine-tuned later if needed
embedding_layer = Embedding(input_dim=vocab_size,
                            output_dim=embedding_dim,
                            weights=[embedding_matrix],
                            input_length=max_seq_len,
                            mask_zero=True, # Important for handling padding
                            trainable=True)(input_layer) # Allow fine-tuning embeddings

# Bidirectional LSTM layer
# return_sequences=True is crucial for sequence labeling
bilstm_layer = Bidirectional(LSTM(units=LSTM_UNITS, return_sequences=True, dropout=DROPOUT_RATE, recurrent_dropout=DROPOUT_RATE))(embed

# TimeDistributed Dense layer
# TimeDistributed Dense layer with softmax activation for standard sequence classification
time_distributed_dense = TimeDistributed(Dense(n_tags))(bilstm_layer)
output_layer = Activation('softmax')(time_distributed_dense)

# Define the model
model = Model(input_layer, output_layer)
model.summary()

# --- 3. Compile Model ---
print("\nCompiling model...")
# Use the CRF potential function as the loss
# The CRF layer handles the decoding and loss calculation internally when used as the output layer.
optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
# Compile with sparse_categorical_crossentropy loss and accuracy metric.
optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])


# --- 4. Train Model ---
print("\nTraining model...")
early_stopping = EarlyStopping(monitor='val_loss', patience=PATIENCE, restore_best_weights=True)
model_checkpoint = ModelCheckpoint(MODEL_SAVE_PATH, monitor='val_loss', save_best_only=True)

history = model.fit(
    X_train, y_train,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping, model_checkpoint]
)

print("Training finished.")
print(f"Best model saved to {MODEL_SAVE_PATH}")

# --- 5. Evaluate Model ---
print("\nEvaluating model on test set...")

# Predict probabilities for each tag at each time step
y_pred_probs = model.predict(X_test)
# Get the tag index with the highest probability at each step
y_pred_indices = np.argmax(y_pred_probs, axis=-1)

# Convert indices back to tags, ignoring padding (where X_test is 0)
y_true_tags = []
y_pred_tags = []

for i in range(len(X_test)):
    true_seq = []
    pred_seq = []
    for j in range(max_seq_len):
        if X_test[i, j] != 0: # Check if it's not a padding token
            true_tag_idx = y_test[i, j]
            pred_tag_idx = y_pred_indices[i, j]
            true_seq.append(idx2tag[true_tag_idx])
            pred_seq.append(idx2tag[pred_tag_idx])
    y_true_tags.append(true_seq)
    y_pred_tags.append(pred_seq)

# Calculate and print classification report
report = classification_report(y_true_tags, y_pred_tags, digits=4)
print("\nClassification Report (Test Set):")
print(report)

# Calculate overall F1 score (micro average is often used for NER)
f1 = f1_score(y_true_tags, y_pred_tags, average='micro')
print(f"\nOverall F1 Score (Micro): {f1:.4f}")
```

```
print("\nEvaluation complete.")
```

Predict LSTM

```
import pickle
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences

# --- Configuration ---
MODEL_PATH = "bilstm_model.h5"
TOKENIZER_PATH = "tokenizer_lstm.pkl"
TAG_MAP_PATH = "tag_map_lstm.pkl"
# Load MAX_SEQ_LEN from the prepared data (or define it if known)
# We'll load it from the prepared data pickle for consistency
PREPARED_DATA_PATH = "prepared_lstm_data.pkl"

# --- Load Model, Tokenizer, Tag Map, and Config ---
print(f"Loading model from {MODEL_PATH}...")
# No custom objects needed as we reverted to standard layers
model = tf.keras.models.load_model(MODEL_PATH)
print("Model loaded successfully.")

print(f"Loading tokenizer from {TOKENIZER_PATH}...")
with open(TOKENIZER_PATH, 'rb') as f:
    word_tokenizer = pickle.load(f)
print("Tokenizer loaded successfully.")

print(f"Loading tag mapping from {TAG_MAP_PATH}...")
with open(TAG_MAP_PATH, 'rb') as f:
    tag_maps = pickle.load(f)
idx2tag = tag_maps['idx2tag']
print("Tag mapping loaded successfully.")

print(f"Loading max sequence length from {PREPARED_DATA_PATH}...")
try:
    with open(PREPARED_DATA_PATH, 'rb') as f:
        prepared_data = pickle.load(f)
    MAX_SEQ_LEN = prepared_data['max_seq_len']
    print(f"Max sequence length set to: {MAX_SEQ_LEN}")
except FileNotFoundError:
    print(f"Error: {PREPARED_DATA_PATH} not found. Using default MAX_SEQ_LEN=50.")
    MAX_SEQ_LEN = 50 # Fallback if the prepared data file is missing
except KeyError:
    print(f"Error: 'max_seq_len' key not found in {PREPARED_DATA_PATH}. Using default MAX_SEQ_LEN=50.")
    MAX_SEQ_LEN = 50 # Fallback if the key is missing

# --- Prediction Function ---
def predict_ner_lstm(sentence, model, tokenizer, idx2tag_map, max_len):
    """Predicts NER tags for a given sentence using the BiLSTM model."""
    # Tokenize the sentence words
    words = sentence.split() # Simple split for this example
    word_sequences = tokenizer.texts_to_sequences([words])

    # Pad the sequence
    padded_sequence = pad_sequences(word_sequences, maxlen=max_len, padding='post')

    # Get model predictions (probabilities)
    pred_probs = model.predict(padded_sequence, verbose=0) # verbose=0 to suppress progress bar

    # Get the tag index with the highest probability for each token
    pred_indices = np.argmax(pred_probs, axis=-1)[0] # Get the predictions for the first (only) sequence

    # Convert indices back to tags
    predicted_tags = [idx2tag_map.get(idx, 'O') for idx in pred_indices[:len(words)]] # Only map tags for original words

    # Combine words and predicted tags
    word_tags = list(zip(words, predicted_tags))

    return word_tags

# --- Example Sentences ---
sentences_to_predict = [
    "Harry Potter went to London to watch the Arsenal game.",
    "Apple announced the new iPhone at the Steve Jobs Theater in Cupertino.",
    "Taylor Swift released her album 'Folklore' last year.",
    "Watching The Office on Netflix is my favorite pastime."
]
```

```
# --- Perform Predictions ---
print("\nPerforming predictions on custom sentences (BiLSTM Model):")
for sentence in sentences_to_predict:
    print(f"\nSentence: {sentence}")
    predicted_tags = predict_ner_lstm(sentence, model, word_tokenizer, idx2tag, MAX_SEQ_LEN)
    print("Predicted Tags:", predicted_tags)

print("\nPrediction complete.")
```