Project Documentation

Aspect-Based Product Review Analysis Engine

Abdulrahman Riyad - 120220193
Abanoub Samy - 120220108
Abdallah Adel - 120220107
Shahd Ammar - 120220098
Yasmeen Sameh - 120220143
CSE 321 - PBL

May 17, 2025

Contents

1 Introduction and Project Goal

This project aims to develop a system capable of analyzing online product reviews to extract specific product features (aspects) mentioned by customers and subsequently determine the sentiment expressed towards these aspects. The ultimate goal is to provide users with a summarized view of a product's pros and cons based on granular, feature-level feedback, thereby aiding in informed purchasing decisions. This document details the Python implementation within a Kaggle Notebook environment, focusing on data preprocessing, model fine-tuning for Aspect Extraction, and evaluation.

2 Development Environment and Libraries

The project is developed within a Kaggle Notebook environment, leveraging its free GPU (NVIDIA P100) resources for model fine-tuning. The primary programming language is Python.

2.1 Step 0: Library Installation and Version Verification

2.1.1 Purpose

To establish a consistent and functional development environment by installing the required Python libraries at specific or minimum compatible versions. This step is crucial for reproducibility and ensuring all necessary tools for data manipulation, deep learning, NLP model handling, and evaluation are available.

2.1.2 Key Operations and Explanation

The initial setup involves installing several key libraries using 'pip':

```
!pip install transformers>=4.42.0 datasets>=2.10.0 torch>=1.9.0 seqeval --quiet
!pip install peft>=0.11.1 --quiet
!pip install accelerate>=0.28.0 --quiet
```

Listing 1: Library Installation Commands

- **`transformers`**: The Hugging Face library providing access to pre-trained Transformer models (like BERT) and tools for fine-tuning. A version greater than or equal to 4.42.0 is targeted to ensure compatibility with features like 'EncoderDecoderCache' and PEFT integrations.
- **`datasets`**: The Hugging Face library for efficiently loading and processing large datasets, with strong integration with 'transformers'. Version 2.10.0 or newer is specified.
- **`torch`**: The PyTorch deep learning framework, which serves as the backend for model computations in this project. Version 1.9.0 or newer is specified.
- **`seqeval`**: A library for evaluating sequence labeling tasks, essential for calculating metrics like precision, recall, and F1-score for Aspect Extraction.
- **`peft`**: The Hugging Face Parameter-Efficient Fine-Tuning library. Although our final model might not use explicit PEFT techniques like LoRA, the 'Trainer' class has internal dependencies or optional integrations with 'peft'. Installing a compatible version helps avoid import errors.
- **`accelerate`**: The Hugging Face library that simplifies running PyTorch scripts across different hardware configurations (CPU, GPU, multi-GPU) and is often used by the 'Trainer' for optimized training.
- The `-quiet` flag is used to minimize verbose installation output.

Following the installation, a verification block confirms the installed versions:

```
print("Libraries installation complete.")
print("\n--- Verifying Versions ---")
try:
    import transformers
    print(f"Transformers version: {transformers.__version__}")
# ... (similar try-except blocks for datasets, torch, peft, accelerate, pandas, numpy) ...
except ImportError: print("Library not found.")
print("--- Verification Complete ---")
```

Listing 2: Version Verification Code

This ensures that the correct libraries are accessible by the Python kernel and reports their active versions.

2.1.3 Outputs

- The specified libraries are installed/updated in the Kaggle session.
- Console output confirming installation and listing the versions of key libraries.

3 Data Loading and Initial Preparation

3.1 Step 1: Loading Raw Data

3.1.1 Purpose

To load the SemEval Aspect-Based Sentiment Analysis (ABSA) datasets for laptop and restaurant reviews from CSV files into pandas DataFrames, combine them, and perform an initial inspection.

3.1.2 Inputs

- `Laptop_Train_v2.csv`: CSV file containing laptop review data.
- `Restaurants_Train_v2.csv`: CSV file containing restaurant review data.
- These files are assumed to live under the Kaggle dataset directory.

3.1.3 Key Operations and Explanation

The code utilizes the ‘pandas‘ library to read and manipulate the data.

```
import pandas as pd
import numpy as np

KAGGLE_INPUT_PATH = "/kaggle/input/sem-eval-absa"
laptop_train_path = f"{KAGGLE_INPUT_PATH}/Laptop_Train_v2.csv"
resto_train_path = f"{KAGGLE_INPUT_PATH}/Restaurants_Train_v2.csv"

print("--- Step 1: Loading Data using pd.read_csv ---")
try:
    df_laptop = pd.read_csv(laptop_train_path, encoding='ISO-8859-1', on_bad_lines='skip')
    df_resto = pd.read_csv(resto_train_path, encoding='ISO-8859-1', on_bad_lines='skip')

    df_laptop['domain'] = 'laptop'
    df_resto['domain'] = 'restaurant'
    df_combined_train = pd.concat([df_laptop, df_resto], ignore_index=True)

```

```
    # Verification prints (info, head)
    print(f"Total combined training instances: {len(df_combined_train)}")
    df_combined_train.info()
    print(df_combined_train.head())
    print("\n--- Dataset Loading Complete ---")
except Exception as e:
    # Error handling
    df_combined_train = None
    print(f"Error: {e}")

if df_combined_train is not None and not df_combined_train.empty:
    print("\nData loading successful.")
# ...
```

Listing 3: Step 1 - Data Loading Code

- **Path Definition**: Full paths to the CSV files are constructed using an f-string and the `KAGGLE_INPUT_PATH`.
- **Reading CSVs**: `pd.read_csv()` is used.
  - `encoding='ISO-8859-1'`: Specifies Latin-1 encoding, which is robust for datasets that might contain special characters not covered by default UTF-8.
  - `on_bad_lines='skip'`: Instructs pandas to skip any lines in the CSV that have formatting errors (e.g., incorrect number of columns), preventing the entire loading process from failing.
- **Adding Domain Column**: A 'domain' column is added to each DataFrame ('df_laptop' and 'df_resto') to label the source of the data before merging. This is crucial for potential domain-specific analysis or if IDs are not globally unique.
- **Concatenation**: `pd.concat([df_laptop, df_resto], ignore_index=True)` merges the two DataFrames vertically. `ignore_index=True` ensures the index of the resulting `df_combined_train` DataFrame is reset to a continuous sequence (0, 1, 2,...).
- **Verification**: `df_combined_train.info()` and `df_combined_train.head()` are used to print a summary of the combined DataFrame (column types, non-null counts, memory usage) and its first few rows, respectively, for initial validation.
- **Error Handling**: A 'try-except' block is used to catch potential 'FileNotFoundError' or other exceptions during loading, printing informative messages.

3.1.4 Outputs

- 'df_laptop': Pandas DataFrame with laptop review data.
- 'df_resto': Pandas DataFrame with restaurant review data.
- 'df_combined_train': A single pandas DataFrame containing all training instances from both domains, with an added 'domain' column. This is the primary output of this step.

3.2 Step 2: Initial Data Cleaning, Standardization, and Type Conversion

3.2.1 Purpose

To prepare the combined dataset (`df_combined_train`) for further processing by standardizing column names, handling missing values in critical fields, ensuring correct data types for numerical columns (character offsets), and performing basic text cleaning.

3.2.2 Inputs

- `df_combined_train`: The pandas DataFrame created in Step 1.

3.2.3 Key Operations and Explanation

```
print("\n--- Step 2: Initial Data Cleaning, Standardization, and Type Conversion ---")
df_cleaned = df_combined_train.copy()

# Standardize column names
df_cleaned.columns = df_cleaned.columns.str.lower().str.replace(' ', '_', regex=False)
essential_cols = ['aspect_term', 'polarity', 'from', 'to', 'sentence']

# Handle Missing Values
df_cleaned.dropna(subset=essential_cols, inplace=True)

# Ensure Correct Numerical Types for 'from' and 'to'
if 'from' in df_cleaned.columns and 'to' in df_cleaned.columns:
    if pd.api.types.is_numeric_dtype(df_cleaned['from']) and pd.api.types.is_numeric_dtype(df_cleaned['to']):
        df_cleaned['from'] = pd.to_numeric(df_cleaned['from'], errors='coerce').fillna(-1).astype(int)
        df_cleaned['to'] = pd.to_numeric(df_cleaned['to'], errors='coerce').fillna(-1).astype(int)
    else: # Attempt coercion if not purely numeric
        df_cleaned['from'] = pd.to_numeric(df_cleaned['from'], errors='coerce').fillna(-1).astype(int)
        df_cleaned['to'] = pd.to_numeric(df_cleaned['to'], errors='coerce').fillna(-1).astype(int)
# ... (Error handling ...

# Basic Text Cleaning
if 'sentence' in df_cleaned.columns and 'aspect_term' in df_cleaned.columns:
    df_cleaned['sentence'] = df_cleaned['sentence'].astype(str).str.strip()
    df_cleaned['aspect_term'] = df_cleaned['aspect_term'].astype(str).str.strip()
# ... (Error handling) ...

# Inspect Polarity Labels
if 'polarity' in df_cleaned.columns:
    unique_polarities = df_cleaned['polarity'].unique()
    print(f"Unique polarity values found: {unique_polarities}")


# Verification
df_cleaned.info()
print(df_cleaned.head())
print("\n--- Initial Cleaning, Standardization, and Type Conversion Complete ---")
```

Listing 4: Step 2 - Data Cleaning Code

- **Copy DataFrame**: `df_cleaned = df_combined_train.copy()` creates a working copy to preserve the original loaded data.
- **Standardize Column Names**: `df_cleaned.columns = df_cleaned.columns.str.lower().str.replace(' ', '_', regex=False)` converts all column names to lowercase and replaces spaces with underscores (e.g., "Aspect Term" becomes "aspect_term"). This ensures consistency and easier programmatic access.
- **Define Essential Columns**: `essential_cols` lists columns critical for ABSA labeling.

- **Handle Missing Values**: `df_cleaned.dropna(subset=essential_cols, inplace=True)` removes rows where any of the `essential_cols` have missing (NaN) values. Rows without these labels are unusable for supervised fine-tuning.
- **Ensure Correct Numerical Types**:
  - The code checks if `from` and `to` columns (character offsets for aspects) are already numeric.
  - `pd.to_numeric(df_cleaned['column'], errors='coerce')`: Attempts to convert the column to a numeric type. If non-numeric values are encountered, they are replaced with `NaN`.
  - `.fillna(-1)`: Any resulting `NaN`s are filled with -1 (a placeholder to allow conversion to `int`).
  - `.astype(int)`: Converts the columns to integer type, which is necessary for using these offsets as indices.
- **Basic Text Cleaning**:
  - `df_cleaned['column'] = df_cleaned['column'].astype(str)`: Ensures the column is treated as string type.
  - `.str.strip()`: Removes leading and trailing whitespace from each entry in the `sentence` and `aspect_term` columns. This prevents issues with tokenization and matching.
- **Inspect Polarity Labels**: `df_cleaned['polarity'].unique()` identifies and prints the distinct sentiment labels present in the dataset (e.g., 'positive', 'negative', 'neutral', 'conflict'). This is important for understanding the sentiment task if pursued.
- **Verification**: `.info()` and `.head()` are called on `df_cleaned` to confirm the changes (correct data types, row counts after NaN removal).

3.2.4 Outputs

- `df_cleaned`: A pandas DataFrame with standardized column names, no missing values in essential columns, correctly typed numerical offset columns ('from', 'to'), and stripped text columns. This DataFrame is the input for the next aggregation step.

3.3 Step 3: Aggregate Aspects per Sentence

3.3.1 Purpose

To transform the data from a "one row per aspect mention" format to a "one row per unique sentence" format. In this new format, each sentence row will contain a list of all aspect details (term, polarity, character offsets) associated with it. This structure is required for the subsequent tokenization and label alignment process.

3.3.2 Inputs

- `df_cleaned`: The pandas DataFrame from Step 2.

3.3.3 Key Operations and Explanation

```
print("\n--- Step 3 (Revised): Aggregate Aspects per Sentence using Unique ID ---")
# Create unique_id
if 'id' in df_cleaned.columns and 'domain' in df_cleaned.columns:
    df_cleaned['unique_id'] = df_cleaned['domain'] + '_' + df_cleaned['id'].astype(
    str)
```

```
    print(f"Unique IDs created: {df_cleaned['unique_id'].nunique()}")
else: # Error handling
    df_cleaned['unique_id'] = None

def aggregate_aspects_per_sentence(group):
    aspect_list = []
    for _, row in group.iterrows():
        aspect_list.append({
            'term': row['aspect_term'], 'polarity': row['polarity'],
            'from': row['from'], 'to': row['to']
        })
    return pd.Series({
        'original_id': group['id'].iloc[0],
        'sentence': group['sentence'].iloc[0],
        'aspects': aspect_list,
        'domain': group['domain'].iloc[0]
    })

if 'unique_id' in df_cleaned.columns and df_cleaned['unique_id'].notna().all():
    aggregated_data_series = df_cleaned.groupby('unique_id').apply(
    aggregate_aspects_per_sentence)
    aggregated_data_series = aggregated_data_series.dropna()
    aggregated_df = aggregated_data_series.reset_index()
    print(f"Number of unique sentences after aggregation: {len(aggregated_df)}")
else: # Error handling
    aggregated_df = pd.DataFrame()

# Verification and Conversion to Hugging Face Dataset
if not aggregated_df.empty:
    print(aggregated_df.head())
    # Example for specific ID (e.g., laptop_3)
    # ...
    from datasets import Dataset
    columns_to_keep = ['unique_id', 'sentence', 'aspects', 'domain']
    hf_dataset = Dataset.from_pandas(aggregated_df[columns_to_keep])
    print(hf_dataset)
    print(hf_dataset[0])
else:
    hf_dataset = None
    print("Aggregation failed or resulted in empty DataFrame.")
print("\n--- Aggregation (Revised) Complete ---")
```

Listing 5: Step 3 - Data Aggregation Code

- **Create Unique Sentence Identifier**:
  - `df_cleaned['unique_id'] = df_cleaned['domain'] + '_' + df_cleaned['id'].astype(str)`: A new column `unique_id` is created by concatenating the `domain` (e.g., "laptop") and the original `id` (converted to string). This ensures a globally unique identifier for each sentence, resolving the issue of duplicate `ids` across different domains.
- **Define `aggregate_aspects_per_sentence` Function**:
  - This function is designed to be applied to each group of rows sharing the same `unique_id`.
  - It iterates through the rows of the input `group` (which is a DataFrame subset for one unique sentence).
  - For each row (representing an aspect mention), it creates a dictionary containing the `term`, `polarity`, `from`, and `to` values.
  - These dictionaries are collected into `aspect_list`.

  - It returns a pandas `Series` containing the `original_id`, the `sentence` text (taken from the first row of the group, as it's constant within the group), the complete `aspect_list`, and the `domain`.

- **Group and Apply**:
  - `aggregated_data_series = df_cleaned.groupby('unique_id').apply(aggregate_aspects_per_sentence)`: The core operation. `df_cleaned` is grouped by the newly created `unique_id`. The `aggregate_aspects_per_sentence` function is then applied to each of these groups.
  - The result `aggregated_data_series` is a pandas Series where the index is the `unique_id`, and each value is the Series returned by the aggregation function.
  - `.dropna()`: Removes any potential `None` rows that might have resulted if a group was skipped in the apply function (due to missing columns).
  - `aggregated_df = aggregated_data_series.reset_index()`: Converts this Series back into a DataFrame, making `unique_id` a regular column and creating a new default integer index.
- **Verification**: Code is included to print the head of `aggregated_df` and a specific example (e.g., the sentence originally with `id=3` from the laptop domain) to ensure the `aspects` list for that sentence is now correctly isolated and contains only its own aspects.
- **Convert to Hugging Face Dataset**:
  - `from datasets import Dataset`: Imports the `Dataset` class from the Hugging Face `datasets` library.
  - `columns_to_keep = ['unique_id', 'sentence', 'aspects', 'domain']`: Specifies which columns are relevant for the next stages.
  - `hf_dataset = Dataset.from_pandas(aggregated_df[columns_to_keep])`: Converts the pandas `aggregated_df` (with selected columns) into a Hugging Face `Dataset` object. This format is optimized for efficient use with `transformers` tokenizers and the `Trainer`.

3.3.4 Outputs

- `aggregated_df`: A pandas DataFrame where each row represents a unique sentence, containing a `unique_id`, the `sentence` text, the `domain`, and an `aspects` column which holds a list of dictionaries, each detailing an aspect mention within that sentence.
- `hf_dataset`: A Hugging Face `Dataset` object derived from `aggregated_df`, containing the `unique_id`, `sentence`, `aspects`, and `domain` columns. This is the primary input for the tokenization and label alignment step.

4 Data Transformation and Model Preparation (Steps 4-7)

This section details the critical phase where the aggregated sentence-level data is transformed into a format suitable for training a Transformer model for Aspect Extraction. It covers tokenization, alignment of aspect labels to tokens, definition of evaluation metrics, configuration of training parameters, and instantiation of the Hugging Face 'Trainer'. These steps were consolidated into a single executable code block to ensure variable scope and state consistency within the Kaggle Notebook environment.

4.1 Initial Imports and Setup

4.1.1 Purpose

To import all necessary libraries and modules at the beginning of the consolidated code block, ensuring all tools are available for the subsequent steps.

4.1.2 Key Operations and Explanation

```
import torch
from transformers import (
    AutoTokenizer,
    AutoConfig,
    DataCollatorForTokenClassification,
    AutoModelForTokenClassification,
    TrainingArguments,
    Trainer
)
from datasets import Dataset, DatasetDict
import pandas as pd
import numpy as np
try:
    from seqeval.metrics import classification_report, accuracy_score
    SEQEVAL_AVAILABLE = True
    print("seqeval imported successfully.")
except ImportError:
    print("Warning: seqeval library not found. Metrics calculation will be basic.")
    SEQEVAL_AVAILABLE = False
    # Define dummy functions if seqeval not found
    def classification_report(y_true, y_pred, output_dict=True, zero_division=0):
    return {}
    def accuracy_score(y_true, y_pred): return 0.0
```

Listing 6: Initial Imports in Consolidated Block

- **`torch`**: Imports the PyTorch library.
- **`transformers`**: Imports several key classes from the Hugging Face Transformers library:
  - ‘AutoTokenizer‘: For loading the appropriate tokenizer for the chosen pre-trained model.
  - ‘AutoConfig‘: For loading and customizing the configuration of the pre-trained model.
  - ‘DataCollatorForTokenClassification‘: A utility to correctly batch and pad token classification data.
  - ‘AutoModelForTokenClassification‘: For loading the pre-trained model architecture with a token classification head.
  - ‘TrainingArguments‘: For specifying all hyperparameters and settings for the training process.
  - ‘Trainer‘: The high-level class that orchestrates the model fine-tuning loop.
- **`datasets`**: Imports ‘Dataset‘ and ‘DatasetDict‘ for working with Hugging Face dataset objects.
- **`pandas`** and **`numpy`**: Standard data manipulation and numerical operation libraries.
- **`seqeval.metrics`**: Attempts to import `classification_report` and `accuracy_score`.
  - `SEQEVAL_AVAILABLE`: A boolean flag is set to track if `seqeval` was successfully imported.

– Dummy functions are defined for `classification_report` and `accuracy_score` if `seqeval` is not found. This prevents runtime errors later if the library is missing, allowing the rest of the pipeline to proceed with basic metrics (though detailed sequence evaluation would be absent).

4.2 Step 4: Tokenization and Label Alignment

4.2.1 Purpose

To convert the textual sentences and their character-span based aspect labels (from `hf_dataset` created in Step 3) into a format suitable for 'bert-base-uncased'. This involves:

1. Defining the BIO (Beginning, Inside, Outside) labeling scheme for Aspect Extraction.
2. Loading the pre-trained 'bert-base-uncased' tokenizer and its configuration.
3. Implementing and applying the function (`tokenize_and_align_labels`) to tokenize each sentence and map the character-level aspect spans to token-level BIO labels.
4. Preparing a data collator for batching and splitting the data into training, validation, and test sets.

4.2.2 Inputs

– `hf_dataset`: The Hugging Face `Dataset` object from Step 3 (containing `unique_id`, `sentence`, `aspects`, `domain`).

– `MODEL_NAME = "bert-base-uncased"`: Specifies the pre-trained model.

4.2.3 Key Operations and Explanation

1. Defining Labeling Scheme and Loading Tokenizer/Config:

```
label_list = ["O", "B-ASP", "I-ASP"]
label2id = {label: i for i, label in enumerate(label_list)}
id2label = {i: label for i, label in enumerate(label_list)}
num_labels = len(label_list)
MODEL_NAME = "bert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
config = AutoConfig.from_pretrained(MODEL_NAME, num_labels=num_labels,
    id2label=id2label, label2id=label2id)
```

Listing 7: Label Scheme and Tokenizer Loading (Step 4)

– `label_list`, `label2id`, `id2label`, `num_labels`: Define the BIO tagging scheme. "O" for Outside an aspect, "B-ASP" for the Beginning of an aspect, and "I-ASP" for Inside an aspect. Mappings between string labels and numerical IDs are created, which the model will use.

– `MODEL_NAME`: Stores the identifier for the pre-trained model.

– `tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)`: Loads the pre-trained tokenizer specifically associated with 'bert-base-uncased'. This tokenizer handles subword tokenization (e.g., WordPiece) and adds special tokens like '[CLS]' and '[SEP]'.

– `config = AutoConfig.from_pretrained(...)`: Loads the pre-trained model's configuration and updates it with our specific task details: `num_labels` (3 for O, B-ASP, I-ASP), and the `id2label` and `label2id` mappings. This config will be used when loading the actual model later to ensure the classification head is correctly sized.

2. The `tokenize_and_align_labels` Function:

```
def tokenize_and_align_labels(examples, tkz, lbl2id, label_all_tokens=False):
    tokenized_inputs = tkz(
        examples["sentence"],
        truncation=True, is_split_into_words=False,
        max_length=512, return_offsets_mapping=True
    )
    all_labels = []
    for i, offset_mapping in enumerate(tokenized_inputs["offset_mapping"]):
        aspects_in_doc = examples["aspects"][i]
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        # Initialize labels: -100 for special, 'O' for others
        label_ids = [-100 if word_id is None else lbl2id["O"] for word_id in
    word_ids]

        for aspect in aspects_in_doc: # For each ground-truth aspect
            asp_start_char = aspect['from']
            asp_end_char = aspect['to']
            token_start_index = -1 # Tracks if we've found the first token of
    this aspect

            for idx, (start_char, end_char) in enumerate(offset_mapping):
                if start_char == end_char == 0: continue # Skip special token
    offsets

                # Check if current token is within the aspect's character span
                if (start_char < asp_end_char) and (end_char > asp_start_char)
    : # Overlap
                    # Only process if not a special token already marked -100
                    if label_ids[idx] != -100:
                        # Logic to determine if this is the first word token
    of the aspect
                        is_first_word_token_of_aspect = True
                        current_token_word_id = word_ids[idx]
                        if current_token_word_id is not None and idx > 0:
                            prev_token_word_id = word_ids[idx-1]
                            # If current token belongs to the same original
    word as previous token
                            if prev_token_word_id == current_token_word_id:
                                is_first_word_token_of_aspect = False

                        # If token's start char aligns with aspect's start
    char & it's the first token of its word
                        if start_char >= asp_start_char and
    is_first_word_token_of_aspect:
                            if label_ids[idx] == lbl2id["O"]: # Prioritize B-
    ASP if not already B/I
                                label_ids[idx] = lbl2id["B-ASP"]
                            token_start_index = idx # Mark that we've started
    an aspect
                        # If token is within aspect but not the identified
    start, or if already B-ASP
                        elif label_ids[idx] == lbl2id["O"]: # Only change if
    it's 'O'
                            label_ids[idx] = lbl2id["I-ASP"]
                            if token_start_index == -1: token_start_index =
    idx

        # Handle subword token labeling strategy
        if not label_all_tokens:
            final_aligned_labels = []
            last_word_id = None
            for idx, word_id in enumerate(word_ids):
```

```
                if word_id is None: final_aligned_labels.append(-100)
                elif word_id == last_word_id: final_aligned_labels.append
    (-100) # Subsequent subword
                else: final_aligned_labels.append(label_ids[idx]) # First
    subword of a word
                last_word_id = word_id
            all_labels.append(final_aligned_labels)
        else:
            all_labels.append(label_ids) # Use all labels if label_all_tokens
    is True

    tokenized_inputs["labels"] = all_labels
    return tokenized_inputs
```

Listing 8: Tokenize and Align Labels Function (Step 4)

- This function processes a batch of `examples` (sentences and their aspects).
- `tokenized_inputs = tkz(...)`: Tokenizes the sentences.
  - `truncation=True`: Truncates sequences longer than `max_length`.
  - `is_split_into_words=False`: Indicates the input `examples["sentence"]` are full strings, not pre-split words.
  - `max_length=512`: Sets the maximum sequence length for BERT.
  - `return_offsets_mapping=True`: Crucially, this requests the tokenizer to return the character start and end offsets for each generated token. This is essential for aligning our character-based aspect spans.
- The code then iterates through each sentence (`i`) and its `offset_mapping` (list of (start_char, end_char) for each token).
- `word_ids = tokenized_inputs.word_ids(batch_index=i)`: Gets a list mapping each token back to its original word index in the sentence. Special tokens like `[CLS]` and `[SEP]` get `None`.
- `label_ids = [-100 if word_id is None else lbl2id["O"] for word_id in word_ids]`: Initializes a list of labels for the current sentence. Special tokens are assigned `-100` (which is ignored by the Hugging Face loss function). All other tokens are initially labeled "O" (Outside).
- The nested loops then iterate through each `aspect` in the current sentence and then through each `token` (with its `start_char`, `end_char` offsets).
- **Alignment Logic**: The core logic within the loops checks if a token's character span `(start_char, end_char)` overlaps with the current aspect's character span `(asp_start_char, asp_end_char)`.
  - If an overlap is found and the token is the beginning of the aspect (determined by checking character offsets and if it's the first token of its original word mapping to the aspect), it's labeled `B-ASP` (if currently 'O').
  - Other overlapping tokens within the aspect are labeled `I-ASP` (if currently 'O').
  - This logic aims to correctly assign BIO tags even for multi-token aspects.
- **Subword Labeling Strategy (`label_all_tokens=False`)**:
  - After initial B/I/O assignment, this part refines labels for subword tokens.

    * If `label_all_tokens` is `False` (which is set in `fn_kwargs`), only the *first* subword token of any given original word retains its B-ASP, I-ASP, or O label.
    * Subsequent subword tokens belonging to the same original word are assigned `-100` so they are ignored during loss calculation and evaluation. This is a common and often effective strategy.

  – `tokenized_inputs["labels"] = all_labels`: Adds the generated list of numerical label sequences to the `tokenized_inputs` dictionary.

3. Applying the Function and Post-Processing:

```
tokenized_dataset = hf_dataset.map(
    tokenize_and_align_labels, batched=True,
    fn_kwargs={'tkz': tokenizer, 'lbl2id': label2id, 'label_all_tokens': False
    },
    remove_columns=hf_dataset.column_names
)

data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)

train_testvalid = tokenized_dataset.train_test_split(test_size=0.2, seed=42)
test_valid = train_testvalid['test'].train_test_split(test_size=0.5, seed=42)
dataset_splits = DatasetDict({
    'train': train_testvalid['train'],
    'validation': test_valid['train'],
    'test': test_valid['test']
})
print(dataset_splits)
```

Listing 9: Applying Tokenization and Data Splitting (Step 4)

  – `tokenized_dataset = hf_dataset.map(...)`: Applies the `tokenize_and_align_labels` function to all examples in `hf_dataset`.
    * `batched=True`: Processes examples in batches for efficiency.
    * `fn_kwargs`: Passes the tokenizer, label map, and `label_all_tokens` flag as additional arguments to the mapping function.
    * `remove_columns=hf_dataset.column_names`: Removes the original columns (like `sentence`, `aspects`) from the dataset, as they are now superseded by the tokenized inputs and numerical labels.

  – `data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)`: Instantiates a data collator. This utility is used by the `Trainer` to dynamically pad sequences within each batch to the length of the longest sequence in that batch. It also handles padding of the `labels` correctly for token classification.

  – `train_test_split()`: The tokenized dataset is split into training (80%), validation (10%), and test (10%) sets using two calls to `train_test_split`. A `seed=42` is used for reproducibility of the splits.

  – `dataset_splits = DatasetDict(...)`: The splits are organized into a `DatasetDict` for convenient access.

4.2.4 Outputs

  – ‘tokenizer‘: The loaded and configured ‘bert-base-uncased‘ tokenizer.

- 'config': The loaded and configured model configuration object for 'bert-base-uncased', updated with label information.
- `label_list`, `label2id`, `id2label`, `num_labels`: Variables defining the BIO labeling scheme.
- `MODEL_NAME`: String "bert-base-uncased".
- `tokenized_dataset`: A Hugging Face 'Dataset' containing the tokenized inputs (`input_ids`, `attention_mask`, `token_type_ids`, `offset_mapping`) and aligned numerical `labels`.
- `data_collator`: An instance of `DataCollatorForTokenClassification`.
- `dataset_splits`: A Hugging Face `DatasetDict` containing `train`, `validation`, and `test` splits of the `tokenized_dataset`. This is the primary data input for the `Trainer`.

4.3 Step 5: Defining Evaluation Metrics

4.3.1 Purpose

To create a function (`compute_metrics`) that calculates relevant performance metrics (precision, recall, F1-score) for the Aspect Extraction (token classification) task, suitable for use with the Hugging Face 'Trainer'.

4.3.2 Inputs

- `id2label`: The dictionary mapping numerical label IDs back to string labels (e.g., 0: "O").
- `SEQEVAL_AVAILABLE`: Boolean flag indicating if `seqeval` was imported.

4.3.3 Key Operations and Explanation

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred # Unpack predictions and true labels
    predictions = np.argmax(predictions, axis=2) # Convert logits to predicted
     label IDs

    true_labels_str = []
    true_predictions_str = []

    # Iterate through each sequence in the batch
    for prediction_seq, label_seq in zip(predictions, labels):
        seq_true_labels = []
        seq_true_preds = []
        for pred_id, label_id in zip(prediction_seq, label_seq):
            if label_id != -100: # Only consider non-ignored labels
                 if label_id in id2label and pred_id in id2label: # Check
    validity
                    seq_true_labels.append(id2label[label_id])
                    seq_true_preds.append(id2label[pred_id])
        if seq_true_labels: # Add if not empty
             true_labels_str.append(seq_true_labels)
             true_predictions_str.append(seq_true_preds)

    if not true_labels_str or not SEQEVAL_AVAILABLE:
        return {"precision": 0.0, "recall": 0.0, "f1": 0.0} # Default if no
    labels or seqeval missing

    results = {}
```

```
    try:
        report = classification_report(
            true_labels_str, true_predictions_str,
            output_dict=True, zero_division=0
        )
        # Extract micro average as overall performance
        results["precision"] = report.get("micro avg", {}).get("precision",
    0.0)
        results["recall"] = report.get("micro avg", {}).get("recall", 0.0)
        results["f1"] = report.get("micro avg", {}).get("f1-score", 0.0)
        if "ASP" in report: # Check if "ASP" entity type exists in report
            results["f1_ASP"] = report["ASP"].get("f1-score", 0.0)
    except Exception as e:
        print(f"Error calculating classification_report: {e}")
        results = {"precision": 0.0, "recall": 0.0, "f1": 0.0}
    return results
```

Listing 10: Compute Metrics Function (Step 5)

- The `compute_metrics` function takes an `EvalPrediction` object (named `eval_pred`) as input, which is provided by the `Trainer` during evaluation. This object contains `predictions` (model's output logits) and `labels` (true label IDs).
- `predictions = np.argmax(predictions, axis=2)`: Converts the raw output logits from the model into predicted label IDs by taking the index of the maximum logit value for each token across the label dimension (axis 2).
- **Label Filtering and Conversion**:
  - The code iterates through each sequence of predictions and true labels.
  - `if label_id != -100`: It crucially filters out any tokens that were assigned the label `-100` (special tokens, subsequent subword tokens). These are not part of the actual evaluation.
  - `id2label[label_id]` and `id2label[pred_id]`: Converts the numerical true label IDs and predicted label IDs back into their string representations (e.g., `"O"`, `"B-ASP"`, `"I-ASP"`), as `seqeval` expects lists of string sequences.
  - `if seq_true_labels:`: Ensures empty label sequences are not passed to `seqeval`.
- **Metric Calculation with `seqeval`**:
  - `report = classification_report(...)`: If `seqeval` is available and there are valid labels, this function from the `seqeval` library is called. It takes the list of true label sequences and the list of predicted label sequences.
  - `output_dict=True`: Returns the report as a Python dictionary.
  - `zero_division=0`: Prevents a `ZeroDivisionError` if a class has no true positives or predicted positives (sets precision/recall/F1 to 0 in such cases).
- **Extracting Results**:
  - The function extracts the overall "micro avg" precision, recall, and F1-score from the `report` dictionary. Micro-averaging gives equal weight to each token, which is often suitable for token classification.
  - It also attempts to extract the F1-score specifically for the "ASP" entity type if present in the report. `seqeval` derives entity-level metrics from the BIO tags.
- Error handling is included for the metric calculation.

4.3.4 Outputs

- `compute_metrics`: A Python function that can be passed to the 'Trainer' to evaluate the model during or after training.

4.4 Step 6: Configure Training Arguments

4.4.1 Purpose

To define all hyperparameters and settings that will control the model fine-tuning process managed by the Hugging Face `Trainer`. Due to persistent environment inconsistencies within Kaggle regarding some `TrainingArguments` parameters (like `evaluation_strategy`), a simplified configuration is used that relies on default behaviors and basic step-based logging/saving.

4.4.2 Inputs

- `dataset_splits`: The `DatasetDict` containing the training data (used to calculate approximate `save_steps`).
- Core hyperparameters like `TRAIN_BATCH_SIZE`, `NUM_EPOCHS`, `LEARNING_RATE`.
- `OUTPUT_DIR`: Path for saving model checkpoints and logs.

4.4.3 Key Operations and Explanation

```
args = None # Initialize
if 'dataset_splits' in locals() and dataset_splits:
    TRAIN_BATCH_SIZE = 16; NUM_EPOCHS = 3; LEARNING_RATE = 2e-5
    OUTPUT_DIR = "/kaggle/working/bert-base-uncased-absa-consolidated"
    # Calculate save_steps to be roughly one epoch
    save_steps_approx = len(dataset_splits['train']) // TRAIN_BATCH_SIZE
    if save_steps_approx == 0: save_steps_approx = 1 # Ensure at least 1

    args = TrainingArguments(
        output_dir=OUTPUT_DIR,
        num_train_epochs=NUM_EPOCHS,
        learning_rate=LEARNING_RATE,
        per_device_train_batch_size=TRAIN_BATCH_SIZE,
        per_device_eval_batch_size=TRAIN_BATCH_SIZE, # Still set for manual eval
        weight_decay=0.01,
        report_to="none", # Disable external reporting
        logging_steps=100, # Log training loss every 100 steps
        save_steps=save_steps_approx, # Save checkpoint ~every epoch
        save_total_limit=2, # Keep only last 2 checkpoints + best (if applicable)
        load_best_model_at_end=False # Disabled due to environment issues
    )
    print("TrainingArguments configured (Simplified Version + Save/Log Steps).")
else:
    print("Error: dataset_splits missing for TrainingArguments.")
```

Listing 11: Simplified Training Arguments (Step 6)

- An `if` condition checks if `dataset_splits` (from Step 4) is available.
- **Hyperparameters**: Basic hyperparameters like `TRAIN_BATCH_SIZE`, `NUM_EPOCHS`, and `LEARNING_RATE` are defined.
- `OUTPUT_DIR`: Specifies the directory for saving outputs (e.g., `/kaggle/working/...`).

- `save_steps_approx`: Calculates the approximate number of steps per epoch to use for `save_steps`, ensuring periodic checkpointing.
- **`TrainingArguments` Instantiation**:
  - `output_dir`, `num_train_epochs`, `learning_rate`, `per_device_train_batch_size`, `per_device_eval_batch_size`, `weight_decay`, `report_to="none"` are standard arguments.
  - `logging_steps=100`: The training loss will be logged every 100 steps.
  - `save_steps=save_steps_approx`: A model checkpoint will be saved approximately every epoch.
  - `save_total_limit=2`: Limits the number of saved checkpoints to save disk space.
  - **`load_best_model_at_end=False`**: This is a key setting adopted due to persistent `TypeError` and `ValueError` issues with Kaggle's environment when trying to use evaluation-dependent arguments like `evaluation_strategy` or `metric_for_best_model`. By setting this to `False`, the `Trainer` will not attempt to automatically identify and load the best model based on validation metrics during training. Evaluation will be performed manually after training.

4.4.4 Outputs

- 'args': An instance of the Hugging Face 'TrainingArguments' class, configured with the simplified set of parameters. This object will control the behavior of the 'Trainer'.

4.5 Step 7: Instantiate the Trainer

4.5.1 Purpose

To create the Hugging Face 'Trainer' object, which encapsulates all the components needed for the fine-tuning process: the model to be trained, training arguments, datasets, data collator, tokenizer, and the evaluation metric computation function.

4.5.2 Inputs

- `MODEL_NAME`: Identifier of the pre-trained model (e.g., "bert-base-uncased").
- 'config': The 'AutoConfig' object loaded in Step 4, updated with label information.
- 'args': The 'TrainingArguments' object configured in Step 6.
- `dataset_splits`: The 'DatasetDict' containing 'train' and 'validation' (and 'test') datasets from Step 4.
- 'tokenizer': The 'AutoTokenizer' loaded in Step 4.
- `data_collator`: The `DataCollatorForTokenClassification` created in Step 4.
- `compute_metrics`: The function defined in Step 5 for evaluating performance.

4.5.3 Key Operations and Explanation

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
trainer = None # Initialize

# Final check of variables before instantiation
```

```
final_check_vars = { # Dictionary for checking
    'model_config': config, 'training_args': args, 'datasets': dataset_splits,
    'tkz': tokenizer, 'collator': data_collator, 'metrics_func':
    compute_metrics
}
if all(v is not None for v in final_check_vars.values()): # Check all are not
    None
    try:
        model = AutoModelForTokenClassification.from_pretrained(MODEL_NAME,
    config=config)
        model.to(device) # Move model to GPU/CPU

        trainer = Trainer(
            model=model,
            args=args,
            train_dataset=dataset_splits["train"],
            eval_dataset=dataset_splits["validation"],
            tokenizer=tokenizer,
            data_collator=data_collator,
            compute_metrics=compute_metrics
        )
        print("Trainer instantiated successfully.")
    except Exception as e: # Catch errors during model loading or Trainer init
        print(f"Error: {e}")
        # traceback.print_exc() # For detailed error
else:
    print("Error: Missing components for Trainer instantiation.")
    # print({k: "Exists" if v is not None else "MISSING/NONE" for k,v in
    final_check_vars.items()})
```

Listing 12: Trainer Instantiation (Step 7)

- `device = torch.device(...)`: Determines if a CUDA-enabled GPU is available and sets the `device` accordingly; otherwise, defaults to CPU.
- `trainer = None`: Initializes `trainer` to `None` so it exists even if instantiation fails.
- **Final Prerequisite Check**: `if all(v is not None for v in final_check_vars.values()):` ensures that all components (`config`, `args`, `dataset_splits`, `tokenizer`, `data_collator`, `compute_metrics`) are defined and not `None` before attempting to create the model and trainer. This was added to debug issues with variable scope in Kaggle.
- **Load Model**:
  - `model = AutoModelForTokenClassification.from_pretrained (MODEL_NAME, config=config)`: Loads the pre-trained `bert-base-uncased` model architecture. Critically, it attaches a new, randomly initialized token classification head suitable for `num_labels` (3 in our case: O, B-ASP, I-ASP), as specified in the `config` object. The weights of the BERT body are pre-trained, while the head's weights will be learned during fine-tuning.
  - `model.to(device)`: Moves the model's parameters and buffers to the selected `device` (GPU if available).
- **Instantiate `Trainer`**:
  - `trainer = Trainer(...)`: Creates an instance of the `Trainer` class.
  - `model=model`: The model to be fine-tuned.
  - `args=args`: The `TrainingArguments` object with all training settings.

    * `train_dataset=dataset_splits["train"]`: The preprocessed training data.
    * `eval_dataset=dataset_splits["validation"]`: The preprocessed validation data. Even if `load_best_model_at_end` is `False` in `args`, providing this allows for manual evaluation calls later (e.g., `trainer.evaluate()`).
    * `tokenizer=tokenizer`: The tokenizer is passed to ensure consistency (e.g., when saving the model, the tokenizer is often saved with it) and for potential internal uses by the `Trainer`.
    * `data_collator=data_collator`: The data collator for dynamic padding.
    * `compute_metrics=compute_metrics`: The function for calculating evaluation metrics.
  - Error handling is included for this critical step.

4.5.4 Outputs

- 'model': The loaded 'bert-base-uncased' model with a token classification head, moved to the appropriate device.
- 'trainer': An instance of the Hugging Face 'Trainer' class, configured and ready to start the fine-tuning process. This is the primary output of this consolidated block for the subsequent training step.
- Console output confirming successful instantiation or detailing errors.

5 Model Fine-Tuning (Step 8)

5.1 Purpose

To execute the training loop using the instantiated Hugging Face 'Trainer' object. This process involves iterating over the training dataset, calculating loss, performing backpropagation to update model weights (primarily the token classification head), and periodically logging training progress and saving model checkpoints.

5.1.1 Inputs

- 'trainer': The Hugging Face 'Trainer' object instantiated in Step 7, fully configured with the model, training arguments, datasets, tokenizer, data collator, and metrics computation function.

5.1.2 Key Operations and Explanation

The core of this step is the invocation of the 'train()' method on the 'trainer' object.

```
print("\n--- Step 8: Start Fine-Tuning ---")
if 'trainer' in locals() and trainer is not None:
    try:
        print("\nStarting training...")
        train_result = trainer.train() # Starts the training loop
        print("\nTraining finished!")

        # Post-training information processing
        metrics = train_result.metrics
        trainer.log_metrics("train", metrics)
        trainer.save_metrics("train", metrics)
        trainer.save_state()
```

```
        print("\nFinal training metrics logged and saved.")
        print(metrics)
    except Exception as e:
        print(f"\nAn error occurred during training: {e}")
else:
    print("Error: Trainer object not found.")
print("\n--- Fine-Tuning Process Attempted ---")
```

Listing 13: Initiating Fine-Tuning (Step 8)

- **Prerequisite Check**: `if 'trainer' in locals() and trainer is not None:` ensures the `trainer` object is available.
- **`train_result = trainer.train()`**: This is the central command. It initiates the training loop managed by the `Trainer`.
  - The `Trainer` iterates through the `train_dataset` for the number of epochs specified in the `TrainingArguments` (`args`).
  - In each step, a batch of data is processed: forward pass through the model, loss calculation, backward pass (gradient computation), and model weight updates via the optimizer.
  - Training loss is logged periodically (based on `logging_steps` in `args`).
  - Model checkpoints are saved periodically (based on `save_steps` in `args`).
- **Post-Training Information**: After `trainer.train()` completes, it returns a `TrainOutput` object (assigned to `train_result`).
  - `metrics = train_result.metrics`: Extracts a dictionary of metrics related to the training run itself (e.g., total training time, average training loss, samples processed per second).
  - `trainer.log_metrics("train", metrics)`: Logs these training metrics to the console and potentially to configured logging integrations.
  - `trainer.save_metrics("train", metrics)`: Saves these training metrics to a JSON file (e.g., `train_results.json`) in the `output_dir`.
  - `trainer.save_state()`: Saves the state of the `Trainer` (including optimizer state, learning rate scheduler state, and RNG states) to the `output_dir`, allowing for resumption of training if needed.
- The final training metrics dictionary is printed to the console.

The success of this step is indicated by a decreasing training loss over iterations and the completion of the specified number of epochs without runtime errors.

5.1.3 Outputs

- Fine-tuned model checkpoints saved to the `output_dir` specified in `TrainingArguments`.
- The final model state saved to `output_dir`.
- `train_results.json` and `trainer_state.json` (and other log files) in `output_dir`.
- The `trainer` object in the Python session now holds the model with its weights updated by the fine-tuning process.

– Console output detailing training progress and final training metrics.

6 Model Evaluation (Step 9)

6.1 Purpose

To assess the performance of the fine-tuned Aspect Extraction model on unseen data. This step uses the validation and test datasets to calculate key metrics such as precision, recall, and F1-score, providing an objective measure of the model's ability to correctly identify aspect terms.

6.1.1 Inputs

– `trainer`: The Hugging Face `Trainer` object, which now contains the fine-tuned model.

– `dataset_splits`: The `DatasetDict` containing the `validation` and `test` data splits, preprocessed in Step 4.

– `compute_metrics`: The function defined in Step 5 for calculating `seqeval` metrics.

6.1.2 Key Operations and Explanation

The evaluation is performed using the 'evaluate()' method of the 'Trainer'.

```
print("\n--- Step 9: Evaluation ---")
if ('trainer' in locals() and trainer is not None and
    'dataset_splits' in locals() and dataset_splits is not None):

    if 'validation' in dataset_splits:
        print("\nEvaluating on the validation set...")
        try:
            eval_results = trainer.evaluate() # Default uses eval_dataset from
     Trainer init
            trainer.log_metrics("eval", eval_results)
            trainer.save_metrics("eval", eval_results)
            print("\nValidation Set Evaluation Results:")
            print(eval_results)
        except Exception as e: print(f"Validation error: {e}")

    if 'test' in dataset_splits:
        print("\nEvaluating on the test set...")
        try:
            test_results = trainer.evaluate(eval_dataset=dataset_splits['test'
    ])
            trainer.log_metrics("test", test_results)
            trainer.save_metrics("test", test_results)
            print("\nTest Set Evaluation Results:")
            # Renaming for clarity in printout
            renamed_test_results = {f"test_{k.replace('eval_', '')}": v for k,
     v in test_results.items()}
            print(renamed_test_results)
        except Exception as e: print(f"Test error: {e}")
else:
    print("Error: Trainer or datasets not ready for evaluation.")
print("\n--- Evaluation Complete ---")
```

Listing 14: Model Evaluation (Step 9)

– **Prerequisite Checks**: Ensures `trainer` and `dataset_splits` (with 'validation' and 'test' keys) are available.

- **Validation Set Evaluation**:
  - `eval_results = trainer.evaluate()`: This command runs the model inference on all examples in `dataset_splits['validation']`. The predictions and true labels are then passed to the `compute_metrics` function.
  - The returned `eval_results` dictionary contains the metrics (e.g., `eval_loss`, `eval_f1`, `eval_precision`, `eval_recall`, and `eval_f1_ASP`).
  - `trainer.log_metrics("eval", eval_results)` and `trainer.save_metrics("eval", eval_results)`: Logs these metrics to the console and saves them to `eval_results.json` in the `output_dir`.
- **Test Set Evaluation (Optional but good practice)**:
  - `test_results = trainer.evaluate(eval_dataset=dataset_splits['test'])`: Explicitly evaluates the model on the `test` set.
  - Metrics are logged and saved similarly, typically to `test_results.json`.
  - The keys in the returned dictionary are often prefixed with `eval_` by default; the code includes a step to rename them to `test_` for clarity in the printout.

The F1-score is a key metric, with observed results of approximately 0.816 on the validation set and 0.838 on the test set, indicating strong performance.

6.1.3 Outputs

- `eval_results`: A dictionary containing performance metrics on the validation set.
- `test_results`: A dictionary containing performance metrics on the test set.
- JSON files (`eval_results.json`, `test_results.json`) containing these metrics saved in the `output_dir`.
- Console output displaying the calculated metrics for both validation and test sets.

7 Saving Model and Inference Pipeline (Step 10)

7.1 Purpose

To save the final fine-tuned model and its corresponding tokenizer to a specified directory for later use (e.g., in a deployment API). This step also demonstrates how to create and use a Hugging Face ‘pipeline‘ for easy inference on new, unseen text using the saved model.

7.1.1 Inputs

- ‘trainer‘: The Hugging Face ‘Trainer‘ object holding the fine-tuned model.
- ‘tokenizer‘: The ‘AutoTokenizer‘ instance used during training.
- ‘id2label‘ (implicitly for pipeline’s understanding of entity groups, though ‘simple‘ aggregation often handles this well).
- `final_model_output_dir`: A string path specifying where to save the model and tokenizer.

7.1.2 Key Operations and Explanation

```
print("\n--- Step 10: Saving Model & Inference Pipeline ---")
from transformers import pipeline # For easy inference

final_model_output_dir = "/kaggle/working/bert-absa-fine-tuned-final"
print(f"\nSaving model and tokenizer to: {final_model_output_dir}")
if 'trainer' in locals() and trainer and 'tokenizer' in locals() and tokenizer:
    trainer.save_model(final_model_output_dir)
    tokenizer.save_pretrained(final_model_output_dir)
    print("Model and tokenizer saved successfully.")
else:
    print("Error: Trainer or Tokenizer not found for saving.")

print("\nCreating inference pipeline...")
if ('AutoModelForTokenClassification' in str(globals().values()) or 'AutoTokenizer' in str(globals().values())): # Simplified check
    try:
        loaded_model = AutoModelForTokenClassification.from_pretrained(final_model_output_dir)
        loaded_tokenizer = AutoTokenizer.from_pretrained(final_model_output_dir)
        device_id = 0 if torch.cuda.is_available() else -1

        absa_pipeline = pipeline(
            "token-classification",
            model=loaded_model,
            tokenizer=loaded_tokenizer,
            aggregation_strategy="simple", # Key for grouping BIO tags
            device=device_id
        )
        print("Inference pipeline created.")

        test_sentences = [
            "The battery life is amazing, but the screen is hard to see.",
            "Great price and camera quality.",
            "Service was slow and the food was just okay."
        ]
        for sentence in test_sentences:
            print(f"\nInput: '{sentence}'")
            outputs = absa_pipeline(sentence)
            extracted_aspects = [
                {"term": entity['word'], "score": entity['score']}
                for entity in outputs if entity['entity_group'] == 'ASP'
            ] # Assuming 'ASP' is the derived entity group
            print(f"Extracted Aspects: {extracted_aspects}")

    except Exception as e: print(f"Error with pipeline: {e}")
else:
    print("Error: Necessary classes for pipeline not available.")
print("\n--- Model Saving and Inference Testing Complete ---")
```

Listing 15: Saving Model and Inference (Step 10)

– **Saving Model and Tokenizer**:

* `final_model_output_dir`: Defines the path for saving.

* `trainer.save_model(final_model_output_dir)`: Saves the model's architecture (in `config.json`) and fine-tuned weights (e.g., `model.safetensors` or `pytorch_model.bin`) to this directory.

    * `tokenizer.save_pretrained(final_model_output_dir)`: Saves the tokenizer's configuration files to the same directory. This ensures the exact tokenizer used for training is packaged with the model.

- **Creating Inference Pipeline**:

    * `from transformers import pipeline`: Imports the `pipeline` utility.

    * `loaded_model = AutoModelForTokenClassification.from_pretrained(final_model_output_dir)`: Loads the fine-tuned model from the specified saved directory.

    * `loaded_tokenizer = AutoTokenizer.from_pretrained(final_model_output_dir)`: Loads the tokenizer from the same directory.

    * `device_id`: Determines if a GPU is available (0 for first GPU, -1 for CPU).

    * `absa_pipeline = pipeline(...)`: Creates a high-level inference pipeline.

        · `"token-classification"`: Specifies the task.

        · `model=loaded_model`, `tokenizer=loaded_tokenizer`: Provides the loaded components.

        · **`aggregation_strategy="simple"`**: This is crucial. For token classification models trained with BIO tags, this strategy automatically groups consecutive `B-ENTITY` and `I-ENTITY` tokens into single entity mentions and attempts to derive an `entity_group` name (e.g., "ASP" from "B-ASP", "I-ASP"). It returns a list of dictionaries, each representing an extracted entity with its text (`word`), confidence score (`score`), and character `start`/`end` offsets.

        · `device=device_id`: Assigns the pipeline to the appropriate device.

- **Testing Inference**:

    * A list of `test_sentences` is defined.

    * The code iterates through these sentences, passing each to `absa_pipeline()`.

    * The output from the pipeline (a list of entity dictionaries) is filtered to keep only entities where `entity['entity_group'] == 'ASP'` (assuming this is how the pipeline names the aspect entities derived from B-ASP/I-ASP tags). The identified aspect `term` and its `score` are then printed. This demonstrates the model's ability to extract aspects from new text.

The successful extraction of terms like "battery life," "screen," "price," etc., with high scores confirms the pipeline's functionality.

7.1.3 Outputs

- A directory (`/kaggle/working/bert-absa-fine-tuned-final`) containing all necessary files for the fine-tuned model and tokenizer.
- `absa_pipeline`: A Hugging Face `pipeline` object ready for performing Aspect Extraction inference.
- Console output showing the extracted aspects and their scores for the provided test sentences.

8 Conclusion

This document has detailed the step-by-step process undertaken to fine-tune a ‘bert-base-uncased‘ model for Aspect Extraction on a combined dataset of laptop and restaurant reviews. The process involved meticulous data loading, cleaning, aggregation, and a complex tokenization and label alignment phase to prepare the data for a token classification task using the BIO scheme. Despite encountering several environment-specific challenges with Kaggle notebooks, particularly concerning library version compatibility and ‘TrainingArguments‘ configuration, these were systematically addressed.

The fine-tuned model achieved promising results, with an F1-score of approximately 0.816 on the validation set and 0.838 on the test set for aspect term extraction. A Hugging Face inference pipeline was successfully created using the saved model, demonstrating its capability to extract relevant aspects from new review sentences.

This project successfully demonstrates the core workflow of fine-tuning a Transformer model for a specific NLP task. Future work will involve integrating this ML model into a larger microservices architecture with web scraping and API components to provide a real-time review analysis tool. Further enhancements could include incorporating sentiment polarity classification for the extracted aspects and exploring more advanced summarization techniques.