NLP Pipeline Project: Hybrid Entity Recognizer for Product Codes

Project Overview

Goal:

The primary objective of this project is to extract and normalize "product codes" (alphanumeric patterns like "AB-1234X") from free-form user reviews and classify each review's overall sentiment.

Steps

1. **Load and Clean Raw Text**:
    - Use regular expressions (regex) to clean and preprocess the raw text data.
2. **Tokenization and POS-Tagging**:
    - Utilize spaCy to tokenize the text and perform part-of-speech (POS) tagging.
    - Develop a custom component within spaCy to recognize product codes based on predefined patterns.
3. **Transformer Input Preparation**:
    - Export the cleaned text to Hugging Face transformer inputs using the `AutoTokenizer`.
4. **Optional Fine-Tuning**:
    - Fine-tune a sentiment classifier on top of the transformer embeddings to improve sentiment analysis accuracy.

Install dependencies

```
!pip install spacy transformers datasets

!python -m spacy download en_core_web_sm

,,
#--------------------------------------------------------------
# Import AutoTokenizer from the Hugging Face Transformers library
#---------------------------------------------------------------
from transformers import AutoTokenizer

# ----------------------------------------------------------------------------------
# Examples with raw text strings: 2 contain product-like codes (e.g. “AB-1234X”),
# ----------------------------------------------------------------------------------
examples = [
    "I bought AB-1234X last week and it works great!",
    "Received product XY-9876 today – totally disappointed.",
    "No code here, just a normal review about quality and price."
]

# ----------------------------------------------------------------------------
# Regex-based cleaner that matches codes like “AB-1234” or “XY-9876Z”
# ----------------------------------------------------------------------------
def clean_and_tag_codes(text: str):
    import re
    PRODUCT_CODE_RE = re.compile(r"\b([A-Z]{2}-\d{4}[A-Z]?)\b")
    codes = PRODUCT_CODE_RE.findall(text)
    cleaned = PRODUCT_CODE_RE.sub("<PRODCODE>", text.lower())
    return cleaned, codes

# ----------------------------------------------------------------------------
# Build cleaned texts: apply clean_and_tag_codes to each string in examples; only
# ----------------------------------------------------------------------------

clean_texts = [clean_and_tag_codes(t)[0] for t in examples]

# ----------------------------------------------------------------------------
# Instantiate tokenizer and encode in one shot: load a pretrained BERT tokenizer.
#--------------------------------------------------------------------

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
encodings = tokenizer(
    clean_texts,
    padding="longest",
    truncation=True,
    max_length=64,
    return_tensors="pt"
)

# ----------------------------------------------------------------------------
# Print the raw tensors - the raw numeric representations that BERT will consume
#---------------------------------------------------

print("Input IDs:\n", encodings.input_ids)
print("Attention Mask:\n", encodings.attention_mask)
```

```
Input IDs:
 tensor([[  101,  1045,  4149, 11113,  1011, 13138,  2549,  2595,  2197,  273
           1998,  2009,  2573,  2307,   999,   102],
        [  101,  2363,  4031,  1060,  2100,  1011,  5818,  2581,  2575,  2651
           1517,  6135,  9364,  1012,   102,     0],
        [  101,  2053,  3642,  2182,  1010,  2074,  1037,  3671,  3319,  2055
           3737,  1998,  3976,  1012,   102,     0]])
Attention Mask:
 tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]])
```

Interpreting `input_ids` & `attention_mask`

`input_ids`

- **Shape**: `(3, 16)` → 3 examples, each padded/truncated to 16 tokens.
- Each number is an index in BERT's vocabulary:
  - **101** = `[CLS]` (start-of-sentence marker)
  - **102** = `[SEP]` (end-of-sentence marker)
  - **0** at the end of rows 2 & 3 = padding token.

`attention_mask`

- **Shape**: `(3, 16)`.
- A `1` means "attend to this token"; `0` means "ignore (padding)".
- Notice that only the first example has 16 real tokens; the other two have a trailing `0`.

```
# Map IDs back to tokens for inspection
print("\n=== Token-level view ===")
for i, ids in enumerate(encodings.input_ids):
    tokens = tokenizer.convert_ids_to_tokens(ids)
    print(f"Example {i} tokens ({len(tokens)}):\n", tokens, "\n")
```

```
=== Token-level view ===
Example 0 tokens (16):
 ['[CLS]', 'i', 'bought', 'ab', '-', '123', '##4', '##x', 'last', 'week', 'an

Example 1 tokens (16):
 ['[CLS]', 'received', 'product', 'x', '##y', '-', '98', '##7', '##6', 'today

Example 2 tokens (16):
 ['[CLS]', 'no', 'code', 'here', ',', 'just', 'a', 'normal', 'review', 'about
```

Regex Code Explanation

The `clean_and_tag_codes` function is a small but critical preprocessing step that helps your model focus on the **pattern** of “there is a product mention here” rather than on the **idiosyncratic** details of each alphanumeric code.

In many real-world NLP pipelines, you’ll often want to **detect**, **extract**, and then **mask** or **tag** certain structured pieces of text—here, “product codes”—for a few key reasons:

1. **Entity Extraction & Downstream Use** By running:

   ```
   codes = PRODUCT_CODE_RE.findall(text)
   ```

   you pull out every substring that looks like a product code (e.g., “AB-1234X”) into a list. You can then store or analyze these codes separately—for instance:

   - Building a lookup table of which exact products occurred in your data.
   - Aggregating statistics per product (average rating, frequency of mention).
   - Linking back to inventory or database records.

2. **Masking to Improve Model Generalization** Raw product codes are essentially arbitrary alphanumeric strings. If you feed them “as-is” into a model (e.g., BERT), the model will see “AB-1234X”, “XY-9876”, etc., each as its own unique token or token sequence. That can:

   - Blow up your vocabulary/embedding usage.

    - Cause the model to “memorize” individual codes instead of learning more general patterns (e.g., “sentiment around a product code”).

   By replacing every code with a single placeholder token (`<PRODCODE>`), you collapse all these variants into one. The model can then learn:

    - “Anytime you see `<PRODCODE>`, treat it like a product mention.”
    - Sentiment or other linguistic cues around codes, without overfitting to the particular alphanumeric patterns.

3. **Privacy & Anonymization** If your product codes are sensitive—say they encode personal data or internal SKUs—you may need to ensure they never leak into model outputs or logs. Masking them with `<PRODCODE>` ensures you have a consistent placeholder without exposing the real codes.

4. **Regex Breakdown**

   ```
   PRODUCT_CODE_RE = re.compile(r"\b([A-Z]{2}-\d{4}[A-Z]?)\b")
   ```

    - `\b`: Word boundary, so we don’t match inside larger words.
    - `[A-Z]{2}`: Exactly two uppercase letters.
    - `-`: A hyphen.
    - `\d{4}`: Exactly four digits.
    - `[A-Z]?`: Optionally one more uppercase letter.
    - The capturing group around it means `.findall()` returns exactly the matched string.

   Examples matched:

    - “AB-1234”
    - “XY-9876Z”

5. **Lower-casing vs. Placeholder Case**

   ```
   cleaned = PRODUCT_CODE_RE.sub("<PRODCODE>", text.lower())
   ```

    - We lowercase the full text so that downstream models (e.g., “bert-base-uncased”) are consistent.
    - We then replace codes with the literal string `"<PRODCODE>"` (already lowercase after `text.lower()`), so they stand out as a single special token.

In summary, any substring that originally looked like AB-1234X (or XY-9876, etc.) will end up being replaced in your text by `<PRODCODE>`.

How to use in a model:

This section demonstrates how to use the preprocessed text with a pre-trained `AutoModelForTokenClassification` from Hugging Face's Transformers library. The model is fine-tuned to recognize product codes in the text.

```
from transformers import AutoModelForTokenClassification

model = AutoModelForTokenClassification.from_pretrained(
    "bert-base-uncased", num_labels=3  # e.g. B-PRODCODE, I-PRODCODE, O
)
outputs = model(
    input_ids=encodings.input_ids,
    attention_mask=encodings.attention_mask
)
# outputs.logits.shape == (3, 16, 3)
```

```
Some weights of BertForTokenClassification were not initialized from the mode
You should probably TRAIN this model on a down-stream task to be able to use
```

Explanation

1. **Model Loading**: The `AutoModelForTokenClassification` is loaded with the `bert-base-uncased` model and configured to recognize three labels: `B-PRODCODE` (beginning of a product code), `I-PRODCODE` (inside a product code), and `O` (outside a product code).
2. **Model Inference**: The model takes the `input_ids` and `attention_mask` tensors as inputs. These tensors are the tokenized and padded/truncated representations of the text.
3. **Output Logits**: The model outputs logits, which are raw prediction scores for each token in the input sequence. The shape of the logits tensor is `(batch_size, sequence_length, num_labels)`, where:
    - `batch_size` is the number of examples in the batch (3 in this case).
    - `sequence_length` is the length of each tokenized sequence (16 in this case).
    - `num_labels` is the number of labels the model is trained to predict (3 in this case).

This setup allows the model to identify and classify each token in the input text, helping to recognize product codes effectively.

Next step: Sentiment analysis

To enhance the functionality of your NLP pipeline, you can include sentiment analysis or fine-tune the model for better performance. By integrating spaCy, you can perform additional NLP tasks such as sentiment analysis, dependency parsing, and named entity recognition, which can complement the product code extraction and improve the overall performance of your pipeline.

Below are the steps to download and use the spaCy model `en_core_web_sm`, which is a “small” English model (≈ 50 MB) containing:

- Tokenizer rules for English
- Part-of-Speech (POS) tagger
- Dependency parser
- Named-Entity Recognizer (NER)

```
# Install necessary libraries
!pip install spacy transformers datasets

# Download the spaCy model
```

```
!python -m spacy download en_core_web_sm

# data_loader.py
from datasets import load_dataset

def load_reviews():
    # e.g. HuggingFace “amazon_polarity” for sentiment
    ds = load_dataset("amazon_polarity", split="train[:1%]")
    # take only text for demo
    return [ex["content"] for ex in ds]

# Load reviews
reviews = load_reviews()
print(reviews[:2])  # Print the first two reviews to verify
```

```
Requirement already satisfied: spacy in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-
Requirement already satisfied: datasets in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: spacy-legacy<3.1.0,>=3.0.11 in /usr/local/lib/p
Requirement already satisfied: spacy-loggers<2.0.0,>=1.0.0 in /usr/local/lib/p
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /usr/local/lib/pyt
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /usr/local/lib/python3.1
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /usr/local/lib/python3
Requirement already satisfied: thinc<8.4.0,>=8.3.4 in /usr/local/lib/python3.1
Requirement already satisfied: wasabi<1.2.0,>=0.9.1 in /usr/local/lib/python3.
Requirement already satisfied: srsly<3.0.0,>=2.4.3 in /usr/local/lib/python3.1
Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in /usr/local/lib/pytho
Requirement already satisfied: weasel<0.5.0,>=0.1.0 in /usr/local/lib/python3.
Requirement already satisfied: typer<1.0.0,>=0.3.0 in /usr/local/lib/python3.1
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /usr/local/lib/python3.1
Requirement already satisfied: numpy>=1.19.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/local/lib/pytho
Requirement already satisfied: pydantic!=1.8,!=1.8.1,<3.0.0,>=1.7.4 in /usr/lo
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/di
Requirement already satisfied: langcodes<4.0.0,>=3.2.0 in /usr/local/lib/pytho
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: huggingface-hub<1.0,>=0.30.0 in /usr/local/lib/
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11
Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.11/di
Requirement already satisfied: dill<0.3.9,>=0.3.0 in /usr/local/lib/python3.11
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: xxhash in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: multiprocess<0.70.17 in /usr/local/lib/python3.
Requirement already satisfied: fsspec<=2025.3.0,>=2023.1.0 in /usr/local/lib/p
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/py
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/py
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in /usr/local/lib/python3.
Requirement already satisfied: language-data>=1.2 in /usr/local/lib/python3.11
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/pyth
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyth
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11
Requirement already satisfied: blis<1.4.0,>=1.3.0 in /usr/local/lib/python3.11
Requirement already satisfied: confection<1.0.0,>=0.0.1 in /usr/local/lib/pyth
Requirement already satisfied: click>=8.0.0 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: shellingham>=1.3.0 in /usr/local/lib/python3.11
Requirement already satisfied: rich>=10.11.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: cloudpathlib<1.0.0,>=0.7.0 in /usr/local/lib/py
Requirement already satisfied: smart-open<8.0.0,>=5.2.1 in /usr/local/lib/pyth
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/di
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dis
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/pytho
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/d
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11/
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.1
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/d
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11/
Requirement already satisfied: marisa-trie>=1.1.0 in /usr/local/lib/python3.11
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/pytho
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-pa
Collecting en-core-web-sm==3.8.0
  Downloading https://github.com/explosion/spacy-models/releases/download/en_c
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.8/12.8 MB 65.4 MB/s eta 0:00:
✔ Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
⚠ Restart to reload dependencies
If you are in a Jupyter or Colab notebook, you may need to restart Python in
order to load all the package's dependencies. You can do this by selecting the
'Restart kernel' or 'Restart runtime' option.
```

```
README.md:   6.81k/? [00:00<00:00, 462kB/s]

train-00000-of-00004.parquet: 100%   260M/260M [00:08<00:00, 9.39MB/s]

train-00001-of-00004.parquet: 100%   258M/258M [00:02<00:00, 108MB/s]

train-00002-of-00004.parquet: 100%   255M/255M [00:04<00:00, 76.7MB/s]
```

```
train-00003-of-                                   254M/254M [00:05<00:00, 75.5MB/s]
00004.parquet: 100%
test-00000-of-                                     117M/117M [00:01<00:00, 123MB/s]
```

Explanation of the Code

Loading the Dataset

```
ds = load_dataset("amazon_polarity", split="train[:1%]")
```

- **Purpose**: Downloads the **Amazon Polarity** dataset, which is a two-class sentiment dataset (positive vs. negative).
- **Details**:
    - `"train[:1%]` means "take the first 1% of the training split." This is for a quick demo; otherwise, the full training set is ~3.6 million examples.
    - `ds` is a `Dataset` object, essentially a list of examples where each example is a dictionary, e.g., `{ "label": 1, "content": "I love this product ..." }`.

Extracting Review Texts

```
return [ex["content"] for ex in ds]
```

- **Purpose**: Builds a plain Python list of strings, extracting only the `"content"` field (the review text).
- **Benefit**: This keeps things simple if you only need raw text.

Loading and Printing Reviews

```
reviews = load_reviews()
print(reviews[:2])
```

- **Purpose**: Calls the loader function to get a list of review strings and prints the first two entries.
- **Benefit**: This allows you to sanity-check that you have successfully loaded the text data.

This setup ensures that you have a list of raw review texts ready for further processing, tokenization, or model training.

Why Replace First, Then Tag:

1. **Normalization**: Lowercasing and other preprocessing steps yield more consistent input for your downstream model (e.g., BERT).
2. **Privacy/Anonymization**: You strip real codes out of the text before training or logging, ensuring sensitive information is protected.
3. **Entity Marking**: By turning each code into `<PRODCODE>`, you give spaCy (and later BERT) an easy hook to spot them, improving the model's ability to recognize product codes.

Transformer Tokenization Demo

You then fed the **cleaned** sentences into BERT’s tokenizer:

- **input_ids**: A tensor of shape `(batch_size, seq_len)` where each integer indexes into BERT’s vocabulary.
- **attention_mask**: Same shape, with `1` for real tokens and `0` for padding.

You also saw how `'<PRODCODE>'` gets split into subwords by BERT’s WordPiece tokenizer.

Key Takeaways

1. **Subword Splitting**
   - Custom tokens like `<PRODCODE>` may be split into subwords by BERT.
   - You need a strategy to align your single-entity label to multiple subwords.
2. **Model Inputs**
   - To fine-tune BERT for token classification (NER), you pass both `input_ids` and `attention_mask` into `AutoModelForTokenClassification`.
   - The model outputs logits of shape `(batch_size, seq_len, num_labels)`.
   - You then decode these logits using `argmax` or a Conditional Random Field (CRF) to get predicted labels for each subword.

Next:

1. **Generate BIO Labels**:
    - From your `<PRODCODE>` spans, create a sequence of `B-PRODCODE`, `I-PRODCODE`, or `O` tags aligned to the tokenized subwords.
2. **Fine-Tune**:
    - Use the `Trainer` API or a custom training loop to fine-tune `AutoModelForTokenClassification` on your silver-labeled data.
3. **Real-Time Processing**:
    - Develop a real-time processing pipeline to extract product codes from live user reviews.
    - Implement a dashboard to visualize the extracted product codes and associated sentiments.
4. **Privacy and Security**:
    - Enhance privacy measures to ensure that sensitive product codes are not exposed.
    - Implement data anonymization techniques to protect user information.
5. **Evaluation and Metrics**:
    - Conduct thorough evaluations using precision, recall, and F1-score metrics.
    - Perform A/B testing to compare the performance of different models and preprocessing techniques.

Final Thoughts

This pipeline demonstrates the power of combining regular expressions, spaCy, and Hugging Face Transformers to build a robust NLP system for product code extraction. The use of BERT for token classification ensures that the model can generalize well to new, unseen data.