To tackle the problem of product deduplication, I propose using a vector database. Since web crawling and scraping can generate large volumes of product data, an efficient method for querying and matching duplicate products is essential. A vector database enables fast similarity searches, making it a suitable choice for this task.

The first step in this solution is to clean the dataset, ensuring that the product attributes are structured and free of inconsistencies. This preprocessing is crucial for obtaining meaningful embeddings.

Once the data is cleaned, we generate embeddings for each product. These embeddings serve as dense vector representations that capture the semantic meaning of product attributes, making it easier to compare products. The embeddings will be stored in the vector database for efficient retrieval.

To detect duplicate products, we perform a nearest-neighbor search on the stored embeddings. For each product, we retrieve its 10 closest neighbors, assuming that the dataset is not large enough to require a higher threshold. We then compute the cosine similarity between the embeddings. If the similarity score indicates a high likelihood of duplication, the products are merged into a single enriched entry.

Another possible solution would be to train a Siamese network, using cosine similarity between product embeddings as a basis for generating labels. This approach could improve the accuracy of duplicate detection. However, it would be impractical in our case, as each newly added product would need to be compared against all existing products, making it computationally expensive.

By leveraging a vector database and embedding-based similarity search, this solution ensures efficient deduplication while maximizing the available product information.

```python
import pandas as pd
import re
```

To generate meaningful embeddings, I selected a subset of product features, specifically the text-based attributes. These fields provide essential context for the embedding model, allowing it to accurately capture the characteristics of each product. By focusing on textual data, we ensure that the model learns a rich semantic representation, improving the effectiveness of similarity searches.

Certain features, such as domain, page URL, and intended industries, could introduce noise into the embeddings, making it harder to match products accurately. Instead, a more advanced approach—given greater computational resources and enhanced web scraping capabilities—would involve extracting product images and combining them with text. This could be achieved using a model like CLIP, which aligns text and image embeddings into a shared space, enabling more robust and precise product matching.

I combined the text data, converted it to lowercase, and removed all special characters that are not alphanumeric.

```python
# Load the Parquet file
parquet_file =
"data/veridion_product_deduplication_challenge.snappy.parquet"
df = pd.read_parquet(parquet_file)

# Define text columns to concatenate
text_columns = ['product_title', 'product_summary', 'product_name',
'brand', 'unspsc', 'description']

# Function to remove punctuation and unknown characters
def remove_punctuation(text):
    return re.sub(r'[^a-zA-Z0-9\s]', '', text)

# Fill missing values and concatenate text fields
df['combined_text'] = df[text_columns].fillna('').apply(
    lambda row: ' '.join(row.astype(str)).strip().lower(), axis=1
)

# Apply punctuation removal
df['combined_text'] = df['combined_text'].apply(remove_punctuation)

# Select relevant columns
df = df[['product_name', 'combined_text']]

# Reset index and add an ID column
df.reset_index(drop=True, inplace=True)
df['id'] = df.index + 1

# Replace empty or whitespace-only product names with "no_name"
df['product_name'] = df['product_name'].apply(lambda x: x.strip() if
isinstance(x, str) and x.strip() else "no_name")

# Reorder columns
df = df[['id', 'product_name', 'combined_text']]

# Save to CSV
csv_file = "data/cleaned_products.csv"
df.to_csv(csv_file, index=False)

print(f"CSV file saved as {csv_file}")
```

```
CSV file saved as data/cleaned_products.csv
```

To generate product embeddings, I use a sentence transformer model that processes the combined textual attributes of each product. Specifically, I use the "all-mpnet-base-v2" model from Sentence Transformers, which produces 768-dimensional embeddings.

The process involves: Tokenizing the text with padding and truncation. Passing it through the transformer model to obtain contextualized token embeddings. Computing a sentence-level embedding by taking a weighted mean of token embeddings, using the attention mask to ignore padding tokens. This approach ensures that each product is represented in a dense vector space, capturing its semantic meaning for effective similarity searches. The generated embeddings are then stored for further processing.

```python
import tensorflow as tf
from transformers import AutoTokenizer, TFAutoModel
import numpy as np
from tqdm import tqdm

# Ensure TensorFlow uses GPU if available
print("Num GPUs Available: ",
len(tf.config.list_physical_devices('GPU')))

# Model for embeddings
model_name = "sentence-transformers/all-mpnet-base-v2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModel.from_pretrained(model_name)
```

```
c:\Users\lucia\miniconda3\envs\tf-gpu\lib\site-packages\tqdm\
auto.py:21: TqdmWarning: IProgress not found. Please update jupyter
and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm

Num GPUs Available:  1

Some weights of the PyTorch model were not used when initializing the
TF 2.0 model TFMPNetModel: ['embeddings.position_ids']
- This IS expected if you are initializing TFMPNetModel from a PyTorch
model trained on another task or with another architecture (e.g.
initializing a TFBertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing TFMPNetModel from a
PyTorch model that you expect to be exactly identical (e.g.
initializing a TFBertForSequenceClassification model from a
BertForSequenceClassification model).
All the weights of TFMPNetModel were initialized from the PyTorch
model.
If your task is similar to the task the model of the checkpoint was
trained on, you can already use TFMPNetModel for predictions without
further training.
```

```python
def get_embedding(text):
    inputs = tokenizer(text, return_tensors="tf", truncation=True,
padding=True)
    outputs = model(**inputs)
    token_embeddings = outputs.last_hidden_state
    attention_mask = tf.cast(inputs["attention_mask"], tf.float32)
    input_mask_expanded = tf.expand_dims(attention_mask, -1)
    sum_embeddings = tf.reduce_sum(token_embeddings *
input_mask_expanded, axis=1)
    sum_mask = tf.reduce_sum(input_mask_expanded, axis=1)
    embedding = sum_embeddings / sum_mask
    return embedding[0].numpy()
```

```
data = pd.read_csv("data/cleaned_products.csv")

# Generate embeddings with progress bar
embeddings = []
for text in tqdm(data["combined_text"], desc="Generating Embeddings",
unit="product"):
    embeddings.append(get_embedding(text).tolist())

# Save embeddings to a new CSV
df_embeddings = pd.DataFrame({"id": data["id"], "product_name":
data["product_name"], "embedding": embeddings})
df_embeddings.to_csv("data/product_embeddings.csv", index=False)
print("Saved embeddings to product_embeddings.csv")
```

```
Generating Embeddings: 100%|██████████| 21946/21946 [1:32:47<00:00,
3.94product/s]

Saved embeddings to product_embeddings.csv
```

The next step is to store the generated embeddings in a vector database for efficient similarity searches. I chose ChromaDB due to its simplicity and ease of use, making it a great option for quickly setting up a product-matching pipeline.

However, other vector databases, such as FAISS, Milvus, or Weaviate, could also be used depending on scalability and performance requirements. The choice of database depends on factors like dataset size, query speed, and integration needs.

```
import chromadb
import ast
import pandas as pd
from tqdm import tqdm

csv_path = "data/product_embeddings.csv"
df = pd.read_csv(csv_path)

# Convert embedding column (assuming it's stored as a string in CSV)
df["embedding"] = df["embedding"].apply(ast.literal_eval)

# Initialize ChromaDB
client = chromadb.PersistentClient(path="chroma_db")
collection = client.get_or_create_collection(
    name="products",
    metadata={"hnsw:space": "cosine"},
)

# Insert products into ChromaDB
collection.add(
    ids=df["id"].astype(str).tolist(),
    embeddings=df["embedding"].tolist(),
    metadatas=[{"product_name": name} for name in df["product_name"]],
)

print("Products stored successfully!")
```

```
Products stored successfully!
```

The final step is to iterate over each product and check its nearest neighbors in the vector database. If a product is identified as a duplicate, it is marked as seen to ensure that we skip it in future iterations (since we have already grouped it with its duplicates).

To determine duplicates, I set a cosine distance threshold of 0.15. Based on initial testing, this threshold strikes a balance between minimizing false negatives (missed duplicates) while avoiding excessive false positives.

For each identified duplicate, we merge their IDs and product names, creating a single enriched entry that consolidates all available information. The final dataset is saved for further analysis and validation.

```
def consolidate_duplicates(threshold=0.15, top_k=10):
    seen = set()
    consolidated = []

    for _, row in tqdm(df.iterrows(), total=len(df), desc="Processing 
products"):
        query_id = str(row["id"])
        query_embedding = row["embedding"]

        if query_id in seen:
            continue  # Skip already processed duplicates

        # Search for similar products
        results = collection.query(
            query_embeddings=[query_embedding],
            n_results=top_k
        )

        matched_ids = results["ids"][0]
        distances = results["distances"][0]

        # Find duplicates within threshold
        duplicates = [query_id]
        for match_id, distance in zip(matched_ids, distances):
            if match_id != query_id and distance < threshold:
                duplicates.append(match_id)
                seen.add(match_id)

        # Merge product ids and names
        merged_ids = " / ".join(duplicates)
        merged_name = " / 
".join(df[df["id"].astype(str).isin(duplicates)]
["product_name"].tolist())

        consolidated.append({
            "ids": merged_ids,
            "product_name": merged_name
        })

    return pd.DataFrame(consolidated)

cleaned_df = consolidate_duplicates()

cleaned_df.to_csv("data/products_output_0_15.csv", index=False)
print("Duplicate consolidation complete! Saved as
'products_output_0_15.csv'.")
```

```
Processing products: 100%|██████████| 21946/21946 [04:37<00:00,
79.21it/s]

Duplicate consolidation complete! Saved as 'products_output_0_15.csv'.
```