SMART PRODUCT REVIEW INSIGHTS USING NLP

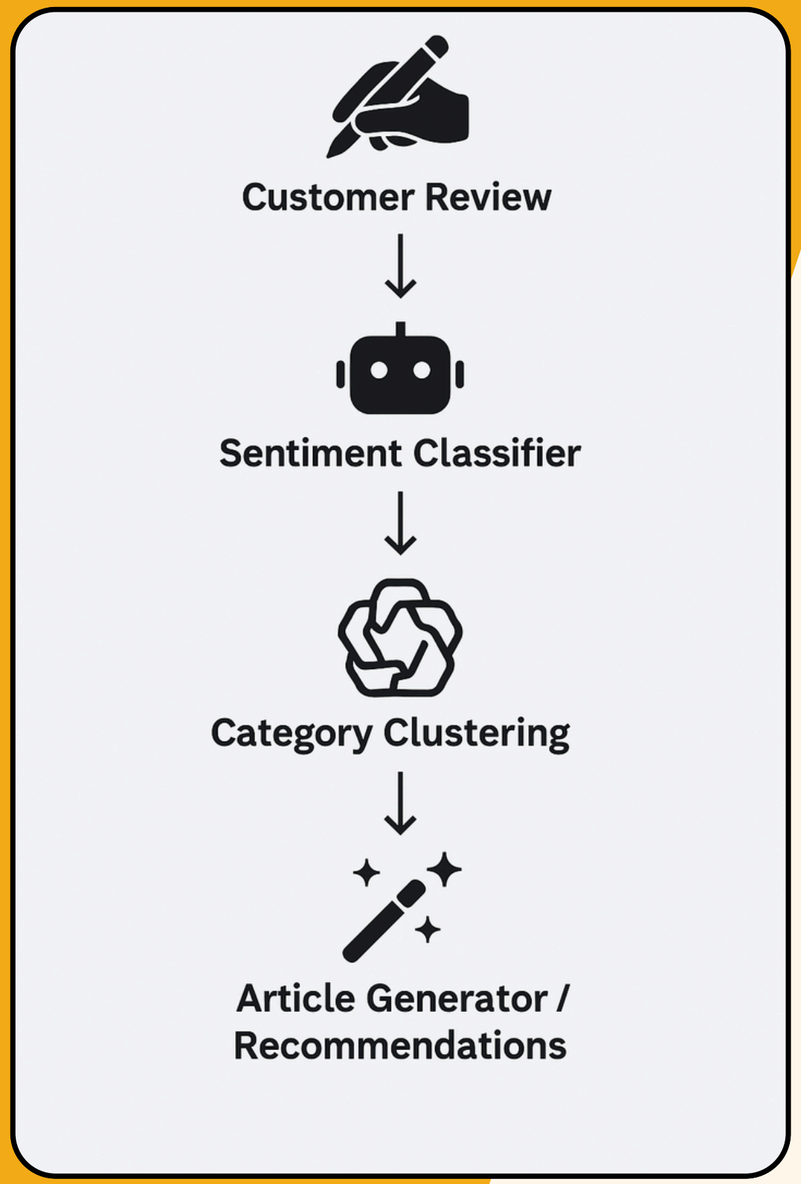
Shatha AlQubaisi

The Problem

Millions of reviews are difficult to analyze manually.

Companies miss out on opportunities to improve their products.

We need a smart tool that summarizes and extracts insights from customer opinions.



My Goal

Build a system that reads and analyzes customer reviews automatically

Provides valuable recommendations for the best and worst product

Powered by advanced NLP models

Amazon Product Reviews Dataset

For this project, I used two versions of the Datafiniti Amazon Product Reviews dataset from Kaggle:

- 🟦 Dataset 1 (May 2019): ~28,000 reviews
- 🟨 Dataset 2 (2017–2018): ~5,000 reviews

Products include Kindle, Fire TV Stick, and other Amazon electronics.
Each review contains key information like: reviews.text, rating, categories, and more.

Task 1
Sentiment Classification

Goal : Classify whether the review is Positive, Negative, or Neutral

Task 0. Preprocessing. Map Star Ratings to Sentiment Classes

```
[ ]  # Import necessary libraries
     import pandas as pd
     import os
     import kagglehub
     import re
```

```
     # Download dataset
     path = kagglehub.dataset_download("datafiniti/consumer-reviews-of-amazon-products")

     # Load both datasets
     df = pd.read_csv(os.path.join(path, "Datafiniti_Amazon_Consumer_Reviews_of_Amazon_Products_May19.csv"))
     df_extra = pd.read_csv(os.path.join(path, "Datafiniti_Amazon_Consumer_Reviews_of_Amazon_Products.csv"))
```

```
[ ]  # Display the first few rows to explore the structure
     df.head()
```

```
[ ]  # Mapping functions
     def map_rating_to_sentiment(star):
         if star in [1, 2]:
             return "Negative"
         elif star == 3:
             return "Neutral"
         elif star in [4, 5]:
             return "Positive"
         return None
```

```
[ ]  label_map = {"Negative": 0, "Neutral": 1, "Positive": 2}
```

Task 1 : Sentiment Classification

```python
def clean_special_symbols(text):
    return re.sub(r"[^\w\s.,!?$%\-]", "", text)

def append_recommendation(row):
    base_text = row['full_text_clean']
    recommend = row['reviews.doRecommend']
    if recommend is True:
        return base_text + " I recommend this product."
    elif recommend is False:
        return base_text + " I do not recommend this product."
    else:
        return base_text

# Preprocessing function
def preprocess(df_input):
    df_input['sentiment'] = df_input['reviews.rating'].apply(map_rating_to_sentiment)
    df_input = df_input.dropna(subset=['sentiment', 'reviews.text'])
    df_input['label'] = df_input['sentiment'].map(label_map)
    df_input['full_text'] = df_input['reviews.title'].fillna('') + ' ' + df_input['reviews.text'].fillna('')
    df_input['full_text_clean'] = df_input['full_text'].apply(clean_special_symbols)
    df_input = df_input[df_input['full_text_clean'].apply(lambda x: len(x.split()) >= 3)].reset_index(drop=True)
    df_input['full_text_enhanced'] = df_input.apply(append_recommendation, axis=1)
    return df_input
```

```python
[ ] # Apply preprocessing to both
    df = preprocess(df)
    df_extra = preprocess(df_extra)
```

Task 1 : Sentiment Classification

Task 1. Classification with pre-trained models.

```python
from sklearn.model_selection import train_test_split

# Step 1: Split 85% for training + validation, and 15% for testing
train_val_texts, test_texts, train_val_labels, test_labels = train_test_split(
    df_balanced['full_text_enhanced'].tolist(),
    df_balanced['label'].tolist(),
    test_size=0.15,
    random_state=42
)

# Step 2: Split the 85% into 70% training and 15% validation
train_texts, val_texts, train_labels, val_labels = train_test_split(
    train_val_texts,
    train_val_labels,
    test_size=0.1765,
    random_state=42
)
```

Task 1 : Sentiment Classification

RoBERTa Model

📌 RoBERTa Model

```
from transformers import RobertaTokenizer, RobertaForSequenceClassification

tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaForSequenceClassification.from_pretrained('roberta-base', num_labels=3)

text = "Replace me by any text you'd like."
encoded_input = tokenizer(text, return_tensors='pt')

output = model(**encoded_input)
logits = output.logits
```

```
[ ] # Tokenize training, validation, and test texts
    train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=128)
    val_encodings = tokenizer(val_texts, truncation=True, padding=True, max_length=128)
    test_encodings = tokenizer(test_texts, truncation=True, padding=True, max_length=128)
```

```
[ ] class ReviewDataset(torch.utils.data.Dataset):
        def __init__(self, encodings, labels):
            self.encodings = encodings
            self.labels = labels

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

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

    # Create datasets
    train_dataset = ReviewDataset(train_encodings, train_labels)
    val_dataset = ReviewDataset(val_encodings, val_labels)
    test_dataset = ReviewDataset(test_encodings, test_labels)
```

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./roberta_results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy"
)
```

Task 1 : Sentiment Classification

RoBERTa Model

```
[ ]  from sklearn.metrics import accuracy_score, f1_score

     def compute_metrics(eval_pred):
         logits, labels = eval_pred
         predictions = logits.argmax(axis=-1)
         return {
             "accuracy": accuracy_score(labels, predictions),
             "f1": f1_score(labels, predictions, average="weighted")
         }
```

```
     from transformers import Trainer

     trainer = Trainer(
         model=model,
         args=training_args,
         train_dataset=train_dataset,
         eval_dataset=val_dataset,
         compute_metrics=compute_metrics,
     )

     trainer.train()
```

```
trainer.train()
```

[7422/7422 27:59, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.403800	0.407901	0.904975	0.859833
2	0.427700	0.406354	0.904975	0.859833
3	0.422900	0.405060	0.904975	0.859833

```
TrainOutput(global_step=7422, training_loss=0.41914859854991915, metr
3904471562374656.0, 'train_loss': 0.41914859854991915, 'epoch': 3.0})
```

```
trainer.evaluate(test_dataset)
```

[531/531 00:28]

```
{'eval_loss': 0.4251943528652191,
 'eval_accuracy': 0.9004951662343786,
 'eval_f1': 0.853347652568013,
 'eval_runtime': 28.121,
 'eval_samples_per_second': 150.813,
 'eval_steps_per_second': 18.883,
 'epoch': 3.0}
```

Task 1 : Sentiment Classification

Oversampling + BERT

Oversampling

```python
from sklearn.utils import resample

# Step 1: Extract relevant columns
df_balance = df[['full_text_enhanced', 'label']]

# Step 2: Split the data by label
df_pos = df_balance[df_balance['label'] == 2]
df_neu = df_balance[df_balance['label'] == 1]
df_neg = df_balance[df_balance['label'] == 0]

# Step 3: Oversample Neutral and Negative to match Positive
df_neu_upsampled = resample(df_neu, replace=True, n_samples=len(df_pos), random_state=42)
df_neg_upsampled = resample(df_neg, replace=True, n_samples=len(df_pos), random_state=42)

# Step 4: Combine all classes
df_balanced = pd.concat([df_pos, df_neu_upsampled, df_neg_upsampled])

# Step 5: Shuffle the dataset
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

# Step 6: Confirm class balance
print(df_balanced['label'].value_counts())
```

```
label
1    25482
2    25482
0    25482
Name: count, dtype: int64
```

📌 BERT Model with Oversampling

In this section, the `bert-base-uncased` model was fine-tuned using only the **May19 dataset**.
To address class imbalance, **oversampling** was applied to the training data.

Model Experiment: BERT (bert-base-uncased)

```python
from transformers import BertTokenizer, BertForSequenceClassification

bert_model_name = "bert-base-uncased"
bert_tokenizer = BertTokenizer.from_pretrained(bert_model_name)
bert_model = BertForSequenceClassification.from_pretrained(bert_model_name, num_labels=3, from_tf=True)
```

```
All TF 2.0 model weights were used when initializing BertForSequenceClassification.

All the weights of BertForSequenceClassification were initialized from the TF 2.0 model.
If your task is similar to the task the model of the checkpoint was trained on, you can already use BertForSeq
```

```python
[ ] bert_train_encodings = bert_tokenizer(train_texts, truncation=True, padding=True, max_length=128)
    bert_val_encodings = bert_tokenizer(val_texts, truncation=True, padding=True, max_length=128)
    bert_test_encodings = bert_tokenizer(test_texts, truncation=True, padding=True, max_length=128)
```

```python
class ReviewDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item["labels"] = torch.tensor(self.labels[idx])
```

Task 1 : Sentiment Classification

Oversampling + BERT

```python
from transformers import Trainer

bert_trainer = Trainer(
    model=bert_model,
    args=bert_training_args,
    train_dataset=bert_train_dataset,
    eval_dataset=bert_val_dataset,
    compute_metrics=compute_metrics,
)

bert_trainer.train()
```

[20067/20067 1:12:53, Epoch 3/3]

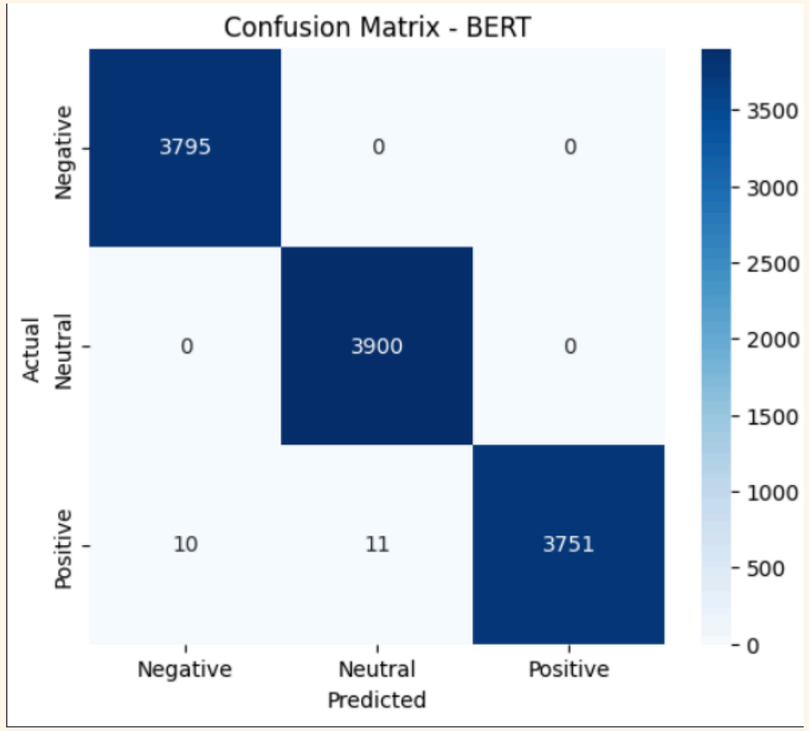
Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.054800	0.027223	0.995204	0.995200
2	0.022000	0.025763	0.995030	0.995025
3	0.001900	0.013585	0.997995	0.997993

```
TrainOutput(global_step=20067, training_loss=0.06740994896509316, metrics=
1.055939923696896e+16, 'train_loss': 0.06740994896509316, 'epoch': 3.0})
```

```python
[ ] bert_trainer.evaluate(bert_test_dataset)
```

[1434/1434 01:23]

```
{'eval_loss': 0.013504719361662865,
 'eval_accuracy': 0.9981686578878521,
 'eval_f1': 0.9981673510350446,
 'eval_runtime': 83.1141,
 'eval_samples_per_second': 137.967,
 'eval_steps_per_second': 17.253,
 'epoch': 3.0}
```



	precision	recall	f1-score	support
Negative	1.00	1.00	1.00	3795
Neutral	1.00	1.00	1.00	3900
Positive	1.00	0.99	1.00	3772
accuracy			1.00	11467
macro avg	1.00	1.00	1.00	11467
weighted avg	1.00	1.00	1.00	11467

Task 1 : Sentiment Classification

DistilBERT + Weighted loss

```
[ ]  # Balancing logic
     current_counts = df['label'].value_counts()
     target_count = current_counts.max()

     needed_counts = {
         0: target_count - current_counts.get(0, 0),
         1: target_count - current_counts.get(1, 0)
     }

     extra_balanced = pd.concat([
         df_extra[df_extra['label'] == 0].sample(n=min(needed_counts[0], df_extra['label'].value_counts().get(0, 0)), random_state=42),
         df_extra[df_extra['label'] == 1].sample(n=min(needed_counts[1], df_extra['label'].value_counts().get(1, 0)), random_state=42)
     ], ignore_index=True)

     df_balanced = pd.concat([df, extra_balanced], ignore_index=True)
```

```
[ ]  # Final label distribution
     print("Final Balanced Distribution:")
     print(df_balanced['label'].value_counts())
```

```
Final Balanced Distribution:
label
2    25482
0     1698
1     1402
Name: count, dtype: int64
```

```
     from sklearn.utils.class_weight import compute_class_weight
     import numpy as np
     import torch

     class_weights = compute_class_weight(
         class_weight="balanced",
         classes=np.unique(df_balanced['label']),
         y=df_balanced['label']
     )

     class_weights_tensor = torch.tensor(class_weights, dtype=torch.float)
```

```
[ ]  from transformers import DistilBertTokenizer, DistilBertForSequenceClassification

     distilbert_model_name = "distilbert-base-uncased"

     distilbert_tokenizer = DistilBertTokenizer.from_pretrained(distilbert_model_name)
     distilbert_model = DistilBertForSequenceClassification.from_pretrained(distilbert_model_name, num_labels=3)
```

```
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-b
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
[ ]  distilbert_train_encodings = distilbert_tokenizer(train_texts, truncation=True, padding=True, max_length=128)
     distilbert_val_encodings = distilbert_tokenizer(val_texts, truncation=True, padding=True, max_length=128)
     distilbert_test_encodings = distilbert_tokenizer(test_texts, truncation=True, padding=True, max_length=128)
```

```
[ ]  distilbert_train_dataset = ReviewDataset(distilbert_train_encodings, train_labels)
     distilbert_val_dataset = ReviewDataset(distilbert_val_encodings, val_labels)
     distilbert_test_dataset = ReviewDataset(distilbert_test_encodings, test_labels)
```

```
     from transformers import TrainingArguments

     distilbert_training_args = TrainingArguments(
         output_dir="./distilbert_results",
         evaluation_strategy="epoch",
         save_strategy="epoch",
         per_device_train_batch_size=8,
         per_device_eval_batch_size=8,
         num_train_epochs=3,
         weight_decay=0.01,
         load_best_model_at_end=True,
         metric_for_best_model="accuracy"
```

Task 1 : Sentiment Classification

DistilBERT + Weighted loss

```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np
import torch

class_weights = compute_class_weight(
    class_weight="balanced",
    classes=np.unique(df_balanced['label']),
    y=df_balanced['label']
)

class_weights_tensor = torch.tensor(class_weights, dtype=torch.float)
```

```
[ ] from transformers import DistilBertTokenizer, DistilBertForSequenceClassification

    distilbert_model_name = "distilbert-base-uncased"

    distilbert_tokenizer = DistilBertTokenizer.from_pretrained(distilbert_model_name)
    distilbert_model = DistilBertForSequenceClassification.from_pretrained(distilbert_model_name, num_labels=3)
```

```
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-b
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
[ ] distilbert_train_encodings = distilbert_tokenizer(train_texts, truncation=True, padding=True, max_length=128)
    distilbert_val_encodings = distilbert_tokenizer(val_texts, truncation=True, padding=True, max_length=128)
    distilbert_test_encodings = distilbert_tokenizer(test_texts, truncation=True, padding=True, max_length=128)
```

```
[ ] distilbert_train_dataset = ReviewDataset(distilbert_train_encodings, train_labels)
    distilbert_val_dataset = ReviewDataset(distilbert_val_encodings, val_labels)
    distilbert_test_dataset = ReviewDataset(distilbert_test_encodings, test_labels)
```

```
from transformers import TrainingArguments

distilbert_training_args = TrainingArguments(
    output_dir="./distilbert_results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy"
```

```
import torch.nn as nn
from transformers import Trainer

# Function to compute weighted loss using class weights
def compute_weighted_loss(logits, labels):
    # Move weights to the same device as the logits
    weights = class_weights_tensor.to(logits.device)
    loss_fn = nn.CrossEntropyLoss(weight=weights)
    return loss_fn(logits, labels)

# Custom Trainer that overrides the default loss computation to include weights
class CustomTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, num_items_in_batch=None):
        labels = inputs.pop("labels")
        outputs = model(**inputs)
        loss = compute_weighted_loss(outputs.logits, labels)
        return (loss, outputs) if return_outputs else loss

# Initialize the CustomTrainer instead of the default Trainer
distilbert_trainer = CustomTrainer(
    model=distilbert_model,
    args=distilbert_training_args,
    train_dataset=distilbert_train_dataset,
    eval_dataset=distilbert_val_dataset,
    compute_metrics=compute_metrics,
)
```

```
[ ] # distilbert_trainer.evaluate(distilbert_test_dataset)
    distilbert_trainer.train()
    distilbert_trainer.evaluate(distilbert_test_dataset)
```

Task 1 : Sentiment Classification

DistilBERT + Weighted loss

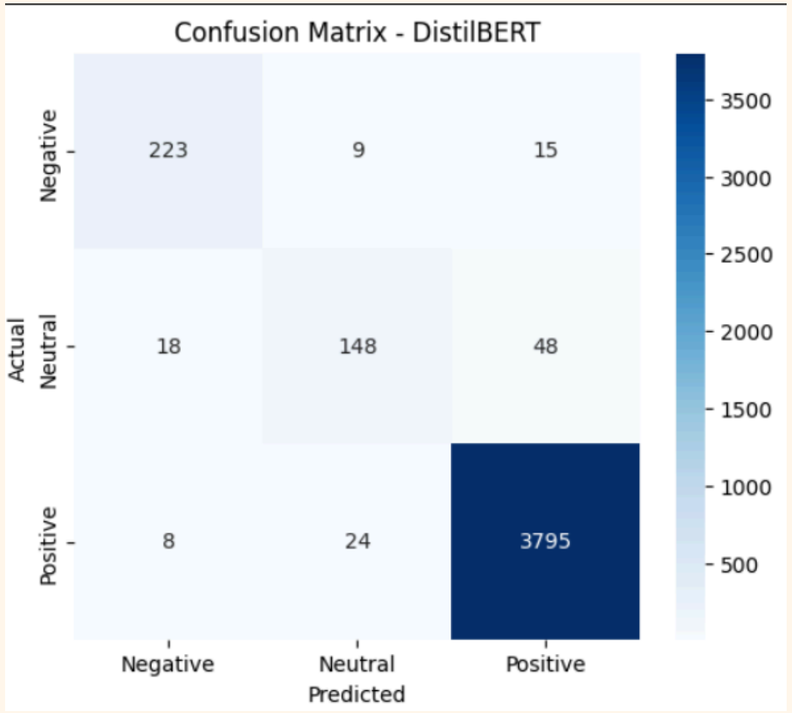
```
# distilbert_trainer.evaluate(distilbert_test_dataset)
distilbert_trainer.train()
distilbert_trainer.evaluate(distilbert_test_dataset)
```

[7503/7503 14:53, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.681200	0.454609	0.958256	0.954820
2	0.360800	0.493572	0.968750	0.968141
3	0.223100	0.509134	0.971549	0.970892

[536/536 00:15]

```
{'eval_loss': 0.5065565705299377,
 'eval_accuracy': 0.9715485074626866,
 'eval_f1': 0.9706522785541777,
 'eval_runtime': 15.2099,
 'eval_samples_per_second': 281.922,
 'eval_steps_per_second': 35.24,
 'epoch': 3.0}
```



	precision	recall	f1-score	support
Negative	0.90	0.90	0.90	247
Neutral	0.82	0.69	0.75	214
Positive	0.98	0.99	0.99	3827
accuracy			0.97	4288
macro avg	0.90	0.86	0.88	4288
weighted avg	0.97	0.97	0.97	4288

Task 1 : Sentiment Classification

DistilBERT + Weighted loss

Manually test for DistilBERT model

```
# Define input
text = "Setup was fine, nothing unusual."
# Tokenize and move to model's device
inputs = distilbert_tokenizer(text, return_tensors="pt")
inputs = {k: v.to(distilbert_model.device) for k, v in inputs.items()}

# Predict
outputs = distilbert_model(**inputs)
prediction = torch.argmax(outputs.logits, dim=1).item()

# Map label
label_map = {0: "Negative", 1: "Neutral", 2: "Positive"}
print("Predicted class:", prediction, "-", label_map[prediction])
```

```
Predicted class: 1 - Neutral
```

Save the best model

```
distilbert_model.save_pretrained("./Final_best_model_distilbert")
distilbert_tokenizer.save_pretrained("./Final_best_model_distilbert")
```

```
('./Final_best_model_distilbert/tokenizer_config.json',
 './Final_best_model_distilbert/special_tokens_map.json',
 './Final_best_model_distilbert/vocab.txt',
 './Final_best_model_distilbert/added_tokens.json')
```

```
[ ] import shutil

    shutil.make_archive("Final_best_model_distilbert", "zip", "Final_best_model_distilbert")
```

```
'/content/Final_best_model_distilbert.zip'
```

Task 2
Product Category Clustering

```python
import pandas as pd
import os
import kagglehub
import re
from sentence_transformers import SentenceTransformer
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from collections import Counter
from sklearn.manifold import TSNE
from umap import UMAP
import seaborn as sns
```

```python
# Download the dataset
path = kagglehub.dataset_download("datafiniti/consumer-reviews-of-amazon-products")

# Load only the May19 dataset
df = pd.read_csv(os.path.join(path, "Datafiniti_Amazon_Consumer_Reviews_of_Amazon_Products_May19.csv"))

# Show shape and first few rows
print("Shape:", df.shape)
df.head()
```

Task 2 : Product Category Clustering

```python
# Combine 'primaryCategories' and 'categories'
df['combined_category'] = df['primaryCategories'].fillna('') + ' ' + df['categories'].fillna('')

# Clean the combined text
def clean_category_text(text):
    if pd.isna(text):
        return ""
    cleaned = re.sub(r"[^a-zA-Z0-9\s]", " ", text)
    cleaned = cleaned.lower()
    cleaned = re.sub(r"\b(aa|aaa)\b", "", cleaned)
    cleaned = re.sub(r"\s+", " ", cleaned).strip()
    return cleaned

df['cleaned_categories'] = df['combined_category'].apply(clean_category_text)

# Remove exact duplicates (after cleaning)
df_cluster = df.drop_duplicates(subset=['cleaned_categories']).reset_index(drop=True)

# Check cleaned data
print("Remaining rows:", len(df_cluster))
```

```
Remaining rows: 61
```

```python
from collections import Counter

all_words = " ".join(df_cluster['cleaned_categories']).split()
word_freq = Counter(all_words)

# Show the 50 most common words
common_words = word_freq.most_common(50)
for word, freq in common_words:
    print(f"{word}: {freq}")
```

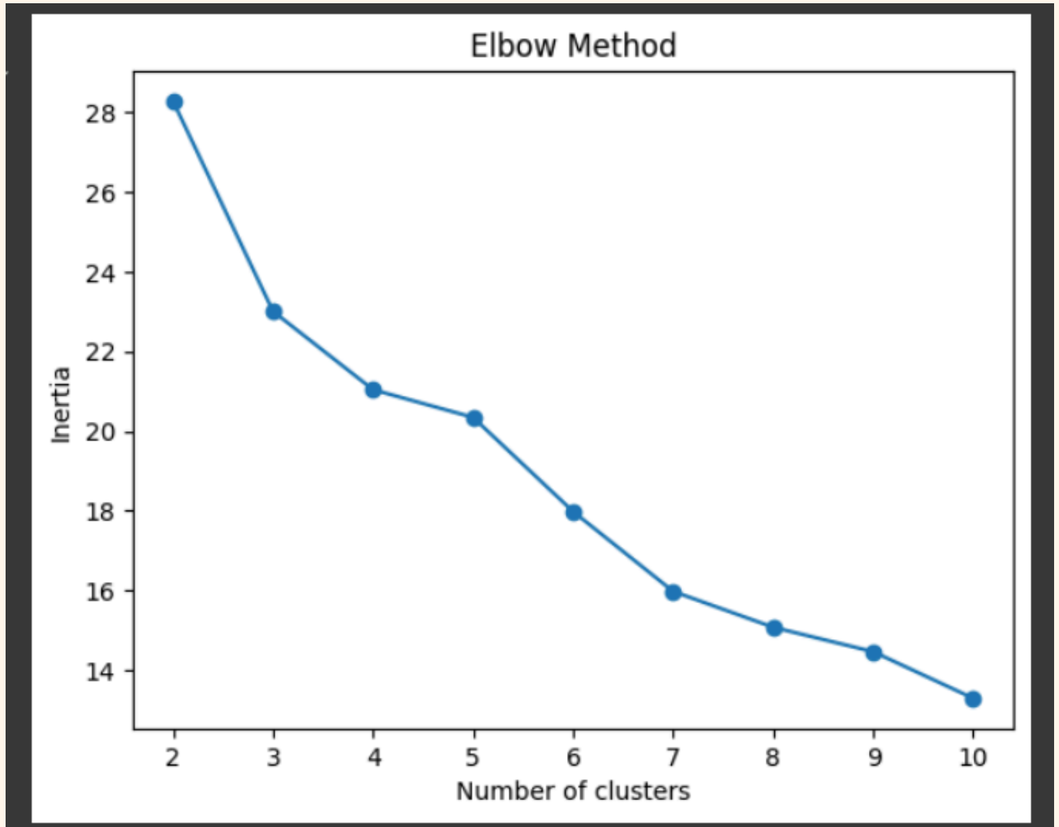
```python
# Remove frequent generic words
words_to_filter = ['electronics', 'features', 'accessories', 'misc', 'miscellaneous', 'device', 'supplies', 'home', 'amazon', 'fire', 'tablet']
def filter_words(text):
    return " ".join([word for word in text.split() if word not in words_to_filter])
df_cluster['cleaned_categories'] = df_cluster['cleaned_categories'].apply(filter_words)
```

Task 2 : Product Category Clustering

```
# Generate category embeddings
embedding_model = SentenceTransformer("all-mpnet-base-v2")
category_embeddings = embedding_model.encode(df_cluster['cleaned_categories'], show_progress_bar=True)
```

Task 2 : Product Category Clustering

```python
# Evaluate number of clusters using Elbow and Silhouette
inertias = []
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(category_embeddings)
    inertias.append(kmeans.inertia_)
plt.plot(range(2, 11), inertias, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method')
plt.show()

for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(category_embeddings)
    score = silhouette_score(category_embeddings, labels)
    print(f"Silhouette Score for k={k}: {score:.4f}")
```




```
Silhouette Score for k=2: 0.1556
Silhouette Score for k=3: 0.2220
Silhouette Score for k=4: 0.1578
Silhouette Score for k=5: 0.1651
Silhouette Score for k=6: 0.1908
Silhouette Score for k=7: 0.2086
Silhouette Score for k=8: 0.1927
Silhouette Score for k=9: 0.1834
Silhouette Score for k=10: 0.2003
```

Task 2 : Product Category Clustering

```
[ ]  # k-means clustering
     kmeans = KMeans(n_clusters=6, random_state=42)
     df_cluster['cluster'] = kmeans.fit_predict(category_embeddings)
```

Cluster quality assessment

```
     print(df_cluster['cluster'].value_counts())
```

```
cluster
2    16
1    16
4    11
5     9
0     6
3     3
Name: count, dtype: int64
```

```
[ ]  # Show sample rows for each cluster
     num_clusters = 6
     for i in range(num_clusters):
         print(f"\n🔸 Cluster {i} samples:")
         print(df_cluster[df_cluster['cluster'] == i]['cleaned_categories'].head(10).to_string(index=False))
```

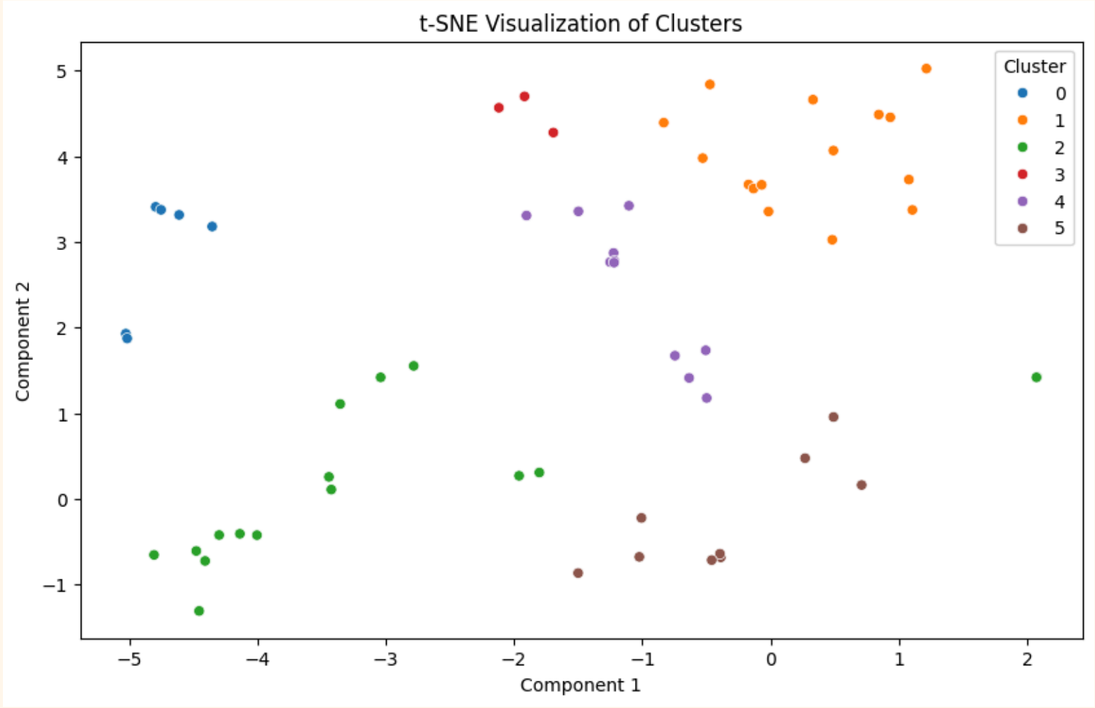
```
animals pet standard litter boxes litter boxes ...
garden garden oven mitts potholders kitchen din...
animals pet carriers totes hard sided carriers ...
garden garden kitchen storage organization kitc...
animals pet crate training crates kennels pet w...
```

```
     # Top words per cluster
     print("\nTop Words per Cluster:")
     for i in range(num_clusters):
         cluster_text = " ".join(df_cluster[df_cluster['cluster'] == i]['cleaned_categories'])
         word_freq = Counter(cluster_text.split())
         top_words = word_freq.most_common(10)
         print(f"\nCluster {i}: {[word for word, _ in top_words]}")
```

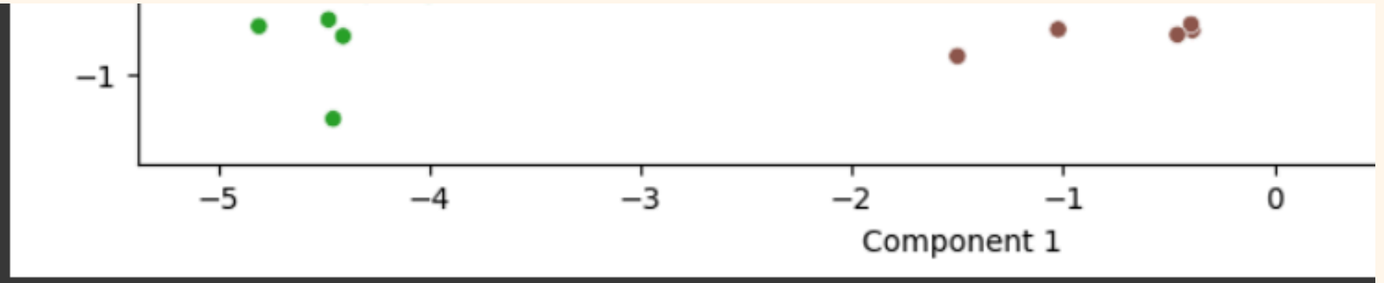
```

Top Words per Cluster:

Cluster 0: ['pet', 'kitchen', 'crates', 'dining', 'animals', 'top', 'products', 'garden', 'kennels', 'dog']
```

Task 2 : Product Category Clustering

```python
# Final: Visualize clusters using t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
tsne_embeddings = tsne.fit_transform(category_embeddings)  # Apply t-SNE on original embeddings

plt.figure(figsize=(10, 6))
sns.scatterplot(x=tsne_embeddings[:, 0], y=tsne_embeddings[:, 1], hue=df_cluster['cluster'], palette='tab10')
plt.title("t-SNE Visualization of Clusters")
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.legend(title="Cluster")
plt.show()

print(df_cluster['cluster'].value_counts())
```



Task 2 : Product Category Clustering




```
cluster
2    16
1    16
4    11
5     9
0     6
3     3
Name: count, dtype: int64
```

```
# Merge similar clusters
merge_map = {0: 0, 1: 1, 2: 2, 3: 3, 4: 3, 5: 4}
df_cluster['merged_cluster'] = df_cluster['cluster'].map(merge_map)
```

Task 2 : Product Category Clustering

Analyze and label the resulting clusters

```
[ ]  # Assign labels to merged clusters
     label_mapping = {
         0: "Home & Pet Supplies",
         1: "Tablets & Kids Tech",
         2: "Smart Home & Audio Devices",
         3: "E-Readers",
         4: "Office & Laptop Accessories"
     }
     df_cluster['cluster_label'] = df_cluster['merged_cluster'].map(label_mapping)
```

```
[ ]  df['cleaned_categories'] = df['cleaned_categories'].apply(filter_words)
```

```
[ ]  df_cluster = df_cluster.drop_duplicates(subset='cleaned_categories')
```

```
[ ]  # Merge labeled clusters back into original dataframe
     df = df.merge(df_cluster[['cleaned_categories', 'cluster_label']], on='cleaned_categories', how='left')
```

```
[ ]  # Create a new column that links each review to its assigned cluster label
     df['review_cluster_label'] = df['cluster_label']

     # Show random samples of reviews for each cluster label to verify the mapping
     for label in df['review_cluster_label'].dropna().unique():
         print(f"\n 🔹 Reviews under: {label}")
         print(df[df['review_cluster_label'] == label][['reviews.text']].dropna().sample(5, random_state=42).to_string(index=False))
```

Task 3
Summarize reviews using generative AI

```
[ ]  import openai
     import os
     import pandas as pd
     from collections import defaultdict
     import markdown
```

```
     Generated code may be subject to a license | pypi.org/project/languru/
     from google.colab import userdata
     openai.api_key = userdata.get("OPENAI_API")
```

```
[ ]  # Clean reviews
     df_reviews = df[['reviews.text', 'cluster_label', 'imageURLs']].dropna(subset=['reviews.text', 'cluster_label'])
     df_reviews['reviews.text'] = df_reviews['reviews.text'].str.strip().str.replace(r'\s+', ' ', regex=True)

     # Group by category
     grouped_reviews = df_reviews.groupby('cluster_label')['reviews.text'].apply(list).to_dict()
     category_summaries = {}
```

Task 3 : Summarize reviews using generative AI

```python
for category, reviews in grouped_reviews.items():
    sample_reviews = reviews[:30] # Take only 30 reviews to adjust the size
    reviews_text = "\n".join(f"- {r}" for r in sample_reviews)

    user_prompt = f"""
Category: {category}

You are a professional blog writer summarizing customer reviews.

Write a blog-style article with the following structure:

1. **Title** — A catchy and relevant blog title
2. **Introduction** — Brief overview of the category and how the products were chosen
3. **Top 3 Recommended Products** — Each with the following format:

Product {1}:
- **Name:** (if mentioned in reviews, otherwise label as Product A/B/C)
- **What makes it stand out:** Summary of positive points
- **Top complaints:** List of common user complaints

(Repeat for Product 2 and 3)

4. **Key Differences** — Compare the 3 products clearly in 2-3 bullet points.

5. **Worst Product to Avoid**
- **Why it's the worst:** Summarize what users complained about

6. **Conclusion** — Give a short helpful wrap-up for buyers.

Use clear markdown formatting and a friendly, informative tone.

Customer Reviews:
{reviews_text}
    """

    try:
        response = openai.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "system", "content": "You are a helpful assistant that writes product recommendation blog posts based on real customer reviews."},
                {"role": "user", "content": user_prompt}
            ],
            temperature=0.7,
            max_tokens=1000
        )

        summary = response.choices[0].message.content
        category_summaries[category] = summary
        print(f"\n 🔹 Article for category: {category}\n{summary}")

    except Exception as e:
        print(f"❌ Error generating article for category {category}: {str(e)}")
```

Task 3 : Summarize reviews using generative AI

```python
from pathlib import Path

# Path to the folder containing all the generated HTML articles
articles_dir = Path("html_articles")
index_path = articles_dir / "index.html"

# HTML header with basic styling
html_header = """
<html>
<head>
    <title>Product Recommendation Blog</title>
    <meta charset="UTF-8">
    <style>
        body {
            font-family: Arial, sans-serif;
            padding: 40px;
            background-color: #f8f8f8;
        }
        h1 {
            color: #333;
        }
        ul {
            list-style-type: none;
            padding: 0;
        }
        li {
            margin: 12px 0;
        }
        a {
            text-decoration: none;
            color: #0066cc;
            font-size: 18px;
        }
        a:hover {
            text-decoration: underline;
        }
    </style>
</head>
<body>
```

```
            color: #0066cc;
            font-size: 18px;
        }
        a:hover {
            text-decoration: underline;
        }
    </style>
</head>
<body>
    <h1>📝 Product Recommendation Articles</h1>
    <ul>
"""

# HTML footer
html_footer = """
    </ul>
</body>
</html>
"""

# Generate the list of links for each article
links_html = ""
for file in sorted(articles_dir.glob("*.html")):
    if file.name == "index.html":
        continue  # Skip the index file itself
    title = file.stem.replace("_", " ").replace("and", "&")
    links_html += f'        <li><a href="{file.name}">📌 {title}</a></li>\n'

# Write the index.html file
with open(index_path, "w", encoding="utf-8") as f:
    f.write(html_header + links_html + html_footer)

print(f"✅ index.html created at: {index_path.absolute()}")
```

```
✅ index.html created at: /content/html_articles/index.html
```

```
[ ] from google.colab import files
    files.download("html_articles/index.html")
```

Task 3 : Summarize reviews using generative AI

Final Output: Generated Blog Articles

E-Readers - Product Recommendation

Blog Title: The Ultimate E-Reader Charger Review: Top Picks and Tips

Introduction

E-readers are a convenient way to carry around your favorite books and documents without the bulk of physical copies. Choosing the right charger for your e-reader can make a big difference in ensuring you always have enough battery life for your reading needs. Based on real customer reviews, we have compiled a list of the top e-reader chargers to help you make an informed decision.

Top 3 Recommended Products

Product 1:

- **Name:** Kindle Fast Charger
- **What makes it stand out:** Charges Kindle devices quickly and efficiently
- **Top complaints:** Some users found that it did not charge as fast as expected, and others wished it came with a longer cord

Deployment – Final App

Built using Streamlit for a simple and interactive UI.

The app allows users to:

- Enter a product review.
- Select a product category.
- Get sentiment prediction (positive, negative, or neutral).
- View a full recommendation article for the selected category.


THANK YOU
FOR YOUR ATTENTION