```
import pandas as pd
import numpy as np
import string
import re
from sklearn.preprocessing import LabelEncoder
import torch
from transformers import DistilBertTokenizer, DistilBertModel
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from google.colab import drive
```

```
from torch.optim import AdamW  # Works for PyTorch >= 1.2.0
```

```
drive.mount('/content/drive')
data_path = '/content/drive/My Drive/restaurants_reviews_dataset'
```

```
Mounted at /content/drive
```

Data Preprocesing

```
df1 = pd.read_csv(data_path + '/Restaurants_Train_v2.csv')
```

```
df1.head()
```

	id	Sentence	Aspect Term	polarity	from	to
0	3121	But the staff was so horrible to us.	staff	negative	8	13
1	2777	To be completely fair, the only redeeming fact...	food	positive	57	61
2	1634	The food is uniformly exceptional, with a very...	food	positive	4	8
3	1634	The food is uniformly exceptional, with a very...	kitchen	positive	55	62
4	1634	The food is uniformly exceptional, with a very...	menu	neutral	141	145

```
df2 = pd.read_csv(data_path + '/Laptop_Train_v2.csv')
```

```
df2.head()
```

	id	Sentence	Aspect Term	polarity	from	to
0	2339	I charge it at night and skip taking the cord ...	cord	neutral	41	45
1	2339	I charge it at night and skip taking the cord ...	battery life	positive	74	86
2	1316	The tech guy then said the service center does...	service center	negative	27	41
3	1316	The tech guy then said the service center does...	"sales" team	negative	109	121
4	1316	The tech guy then said the service center does...	tech guy	neutral	4	12

```
df = pd.concat([df1, df2])
```

```
df.isna().sum()
```

	0
id	0
Sentence	0
Aspect Term	0
polarity	0
from	0
to	0

```
df["polarity"].value_counts()
```

polarity	count
positive	3151
negative	1671
neutral	1093
conflict	136

Cleaning Reviews Text

```
def clean_text(text):
  """Clean and preprocess text data
  Parameters:
  -----------
  text : str
  The text to clean

  Returns:
  --------
  str
  Cleaned text
  """

  # Convert to lowercase
  text = text.lower()

  # Handle contractions
  contractions = {
      "isn't": "is not", "aren't": "are not", "wasn't": "was not", "weren't": "were not","haven't": "have not",
      "hasn't": "has not", "hadn't": "had not", "doesn't": "does not", "don't": "do not", "didn't": "did not",
      "won't": "will not", "wouldn't": "would not", "can't": "cannot", "couldn't": "could not", "shouldn't": "should not",
      "mightn't": "might not", "mustn't": "must not", "i'm": "i am", "you're": "you are", "he's": "he is", "she's": "she is",
      "it's": "it is", "we're": "we are", "they're": "they are", "i've": "i have", "you've": "you have", "we've": "we have",
      "they've": "they have", "i'd": "i would", "you'd": "you would", "he'd": "he would", "she'd": "she would", "it'd": "it would",
      "we'd": "we would", "they'd": "they would", "i'll": "i will", "you'll": "you will", "he'll": "he will", "she'll": "she will",
      "it'll": "it will", "we'll": "we will", "they'll": "they will", "didnt": "did not", "dont": "do not", "cant": "cannot", "wont": "
      }

  for contraction, expansion in contractions.items():
    text = text.replace(contraction, expansion)

  # Preserve emotions
  emoticons = {
      ':)': ' HAPPY_FACE ',
      ':(': ' SAD_FACE ',
      ':D': ' LAUGH_FACE ',
      ':/': ' CONFUSED_FACE '
      }
  for emoticon, replacement in emoticons.items():
    text = text.replace(emoticon, replacement)


  # Remove punctuation but preserve sentence structure
  text = re.sub(f'[{re.escape(string.punctuation)}]', ' ', text)

  # Remove extra whitespace
  text = re.sub(r'\s+', ' ', text).strip()

  # Restore emotions
  for placeholder, emoticon in {v: k for k, v in emoticons.items()}.items():
    text = text.replace(placeholder, emoticon)

  return text
```

```
cleaned_df = df.copy()
cleaned_df['Sentence'] = cleaned_df['Sentence'].astype(str).apply(clean_text)
```

```
cleaned_df.head()
```

	id	Sentence	Aspect Term	polarity	from	to
0	3121	but the staff was so horrible to us	staff	negative	8	13
1	2777	to be completely fair the only redeeming facto...	food	positive	57	61
2	1634	the food is uniformly exceptional with a very ...	food	positive	4	8
3	1634	the food is uniformly exceptional with a very ...	kitchen	positive	55	62
4	1634	the food is uniformly exceptional with a very	menu	neutral	141	145

Encoding Polarity Column

```
sentiment_encoder = LabelEncoder()

cleaned_df['polarity'] = sentiment_encoder.fit_transform(cleaned_df['polarity'])

# Print the mapping of labels to numbers
label_mapping = dict(zip(sentiment_encoder.classes_, range(len(sentiment_encoder.classes_))))
print(label_mapping)
```

```
{'conflict': 0, 'negative': 1, 'neutral': 2, 'positive': 3}
```

```
cleaned_df.head()
```

	id	Sentence	Aspect Term	polarity	from	to
0	3121	but the staff was so horrible to us	staff	1	8	13
1	2777	to be completely fair the only redeeming facto...	food	3	57	61
2	1634	the food is uniformly exceptional with a very ...	food	3	4	8
3	1634	the food is uniformly exceptional with a very ...	kitchen	3	55	62
4	1634	the food is uniformly exceptional with a very	menu	2	141	145

```
cleaned_df['polarity'].dtype
```

```
dtype('int64')
```

```
cleaned_df = cleaned_df.drop(['id', 'from', 'to'], axis=1)
```

```
cleaned_df.head()
```

	Sentence	Aspect Term	polarity
0	but the staff was so horrible to us	staff	1
1	to be completely fair the only redeeming facto...	food	3
2	the food is uniformly exceptional with a very ...	food	3
3	the food is uniformly exceptional with a very ...	kitchen	3
4	the food is uniformly exceptional with a very	menu	2

DistilBERT Fine-tuning

```
# Tokenizer setup
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Goog
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
```

tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 2.81kB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 656kB/s]

tokenizer.json: 100% 466k/466k [00:00<00:00, 2.61MB/s]

config.json: 100% 483/483 [00:00<00:00, 41.0kB/s]

Dataset Setup

```
class AspectSentimentDataset(Dataset):
    def __init__(self, dataframe, tokenizer, max_len=128):
        self.data = dataframe.reset_index(drop=True)
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, idx):
        review = str(self.data.loc[idx, "text"])
        aspect = str(self.data.loc[idx, "aspect"])
        label = int(self.data.loc[idx, "label"])

        encoded = self.tokenizer(
            review,
            aspect,
            truncation=True,
            padding="max_length",
            max_length=self.max_len,
            return_tensors="pt"
        )

        item = {
            "input_ids": encoded["input_ids"].squeeze(0),
            "attention_mask": encoded["attention_mask"].squeeze(0),
            "label": torch.tensor(label, dtype=torch.long)
        }

        return item
```

```
train_df, temp_df = train_test_split(cleaned_df, test_size=0.3, random_state=42)

val_df, test_df = train_test_split(temp_df, test_size=0.5, random_state=42)

# Verify sizes
print(f"Train: {len(train_df)} samples ({len(train_df)/len(cleaned_df)*100:.1f}%)")
print(f"Val: {len(val_df)} samples ({len(val_df)/len(cleaned_df)*100:.1f}%)")
print(f"Test: {len(test_df)} samples ({len(test_df)/len(cleaned_df)*100:.1f}%)")

# Create datasets
train_dataset = AspectDataset(train_df['Sentence'].values, train_df['Aspect Term'].values, train_df['polarity'].values)
val_dataset = AspectDataset(val_df['Sentence'].values, val_df['Aspect Term'].values, val_df['polarity'].values)
test_dataset = AspectDataset(test_df['Sentence'].values, test_df['Aspect Term'].values, test_df['polarity'].values)
```

```
Train: 4235 samples (70.0%)
Val: 908 samples (15.0%)
Test: 908 samples (15.0%)
```

Model 1

Model Architecture

Intermediate layers usually hurt the performance when the dataset is small

```
# Model Architecture
class AspectSentimentClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        self.dropout = torch.nn.Dropout(0.2)

        # Enhanced classifier with intermediate layers
        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(768 * 2, 512),  # Intermediate layer 1
            torch.nn.ReLU(),
            torch.nn.LayerNorm(512),
            torch.nn.Linear(512, 128),      # Intermediate layer 2
            torch.nn.GELU(),
            torch.nn.Linear(128, 4)         # Final output (4 classes)
        )

    def forward(self, sentence_input_ids, sentence_attention_mask, aspect_input_ids, aspect_attention_mask):
        # Get sentence embeddings
        sentence_outputs = self.bert(
            input_ids=sentence_input_ids,
            attention_mask=sentence_attention_mask
        )
        sentence_embedding = sentence_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Get aspect embeddings
        aspect_outputs = self.bert(
            input_ids=aspect_input_ids,
            attention_mask=aspect_attention_mask
        )
        aspect_embedding = aspect_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Combine features
        combined = torch.cat([sentence_embedding, aspect_embedding], dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)

        return logits
```

Training

```
# Training Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = AspectSentimentClassifier().to(device)
optimizer = AdamW(model.parameters(), lr=2e-5)
criterion = torch.nn.CrossEntropyLoss()

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
test_loader = DataLoader(test_dataset, batch_size=16)
```

```
# Training Loop
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss = 0

    for batch in dataloader:
        optimizer.zero_grad()

        inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
        labels = batch['labels'].to(device)

        outputs = model(**inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)

def evaluate(model, dataloader, criterion, debug=False):
    model.eval()
```

```python
    total_loss = 0.0
    predictions, true_labels = [], []
    class_losses = [0.0] * 4
    class_counts = [0] * 4

    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
            labels = batch['labels'].to(device)

            # Forward pass
            logits = model(**inputs)
            loss = criterion(logits, labels)
            total_loss += loss.item()

            # Per-class loss calculation
            for class_idx in range(4):
                mask = (labels == class_idx)
                if mask.any():
                    class_loss = criterion(logits[mask], labels[mask]).item()
                    class_losses[class_idx] += class_loss * mask.sum().item()
                    class_counts[class_idx] += mask.sum().item()

                    if debug:
                        print(f"Batch {batch_idx} Class {class_idx} Loss: {class_loss:.4f} (n={mask.sum().item()})")

            # Store predictions
            _, preds = torch.max(logits, dim=1)
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    # Calculate metrics
    avg_loss = total_loss / len(dataloader)
    class_avg_losses = [
        class_losses[i] / class_counts[i] if class_counts[i] > 0 else 0.0
        for i in range(4)
    ]

    return {
        'loss': avg_loss,
        'class_losses': class_avg_losses,
        'class_counts': class_counts,
        'accuracy': accuracy_score(true_labels, predictions),
        'f1': f1_score(true_labels, predictions, average='macro')
    }
```

```python
best_val_loss = float('inf')

# Training Loop
for epoch in range(10):
    model.train()
    train_loss = train_epoch(model, train_loader, optimizer, criterion)

    model.eval()
    val_results = evaluate(model, val_loader, criterion, debug=(epoch == 0))

    # Improved printing
    print(f"\nEpoch {epoch + 1}")
    print(f"Train Loss: {train_loss:.4f}")
    print(f"Val Loss: {val_results['loss']:.4f}")
    print(f"Val Accuracy: {val_results['accuracy']:.4f}")
    print(f"Val F1: {val_results['f1']:.4f}")

    # Class-wise performance
    class_names = ['conflict', 'negative', 'neutral', 'positive']
    print("\nClass-wise Val Loss:")
    for i, name in enumerate(class_names):
        if val_results['class_counts'][i] > 0:
            print(f"{name.upper():<9}: {val_results['class_losses'][i]:.4f} (n={val_results['class_counts'][i]})")
    print("-----")

    # Save the best model based on validation loss (lower is better)
    if val_results['loss'] < best_val_loss:
        best_val_loss = val_results['loss']
```

```
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': train_loss,
            'val_loss': val_results['loss'],
            'val_accuracy': val_results['accuracy'],
        }, 'best_model1.pth')
        print(f"Saved new best model with val loss: {best_val_loss:.4f}")
```

```
Class-wise Val Loss:
CONFLICT : 2.1132 (n=15)
NEGATIVE : 0.9379 (n=245)
NEUTRAL  : 2.0348 (n=173)
POSITIVE : 0.3417 (n=475)
-----

Epoch 7
Train Loss: 0.1887
Val Loss: 0.8970
Val Accuracy: 0.7500
Val F1: 0.5803

Class-wise Val Loss:
CONFLICT : 2.6902 (n=15)
NEGATIVE : 1.0048 (n=245)
NEUTRAL  : 1.7662 (n=173)
POSITIVE : 0.4692 (n=475)
-----

Epoch 8
Train Loss: 0.1528
Val Loss: 1.0045
Val Accuracy: 0.7368
Val F1: 0.5705

Class-wise Val Loss:
CONFLICT : 3.5712 (n=15)
NEGATIVE : 1.2186 (n=245)
NEUTRAL  : 1.9856 (n=173)
POSITIVE : 0.4568 (n=475)
-----

Epoch 9
Train Loss: 0.1226
Val Loss: 1.0020
Val Accuracy: 0.7280
Val F1: 0.5801

Class-wise Val Loss:
CONFLICT : 2.3535 (n=15)
NEGATIVE : 1.1660 (n=245)
NEUTRAL  : 1.6199 (n=173)
POSITIVE : 0.6539 (n=475)
-----

Epoch 10
Train Loss: 0.1112
Val Loss: 1.1271
Val Accuracy: 0.7434
Val F1: 0.5767

Class-wise Val Loss:
CONFLICT : 3.5132 (n=15)
NEGATIVE : 1.5439 (n=245)
NEUTRAL  : 2.0003 (n=173)
POSITIVE : 0.5235 (n=475)
-----
```

```
"""
best_model = torch.load('best_model1.pth')
model.load_state_dict(best_model['model_state_dict'])"""

checkpoint = torch.load('best_model1.pth')
print(f"Model 1 Best validation loss: {checkpoint['val_loss']:.4f}")
print(f"Achieved at epoch: {checkpoint['epoch'] + 1}")

# Evaluate on the test set ONLY ONCE
model.eval()
test_results = evaluate(model, test_loader, criterion)
print(f"\nModel 1 Final Test Performance:")
print(f"Test Loss: {test_results['loss']:.4f}")
print(f"Test Accuracy: {test_results['accuracy']:.4f}")
print(f"Test F1: {test_results['f1']:.4f}")
```

```
Model 1 Best validation loss: 0.6287
Achieved at epoch: 2

Model 1 Final Test Performance:
Test Loss: 1.0029
Test Accuracy: 0.7335
Test F1: 0.5139
```

```
# save model
torch.save(model.state_dict(), 'model1.pt')
```

Model 2

Model Architecture

```
class AspectSentimentClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        self.dropout = torch.nn.Dropout(0.2)

        # Enhanced classifier with intermediate layers
        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(768 * 2, 512),  # Intermediate layer 1
            torch.nn.ReLU(),
            torch.nn.LayerNorm(512),
            torch.nn.Linear(512, 128),      # Intermediate layer 2
            torch.nn.GELU(),
            torch.nn.Linear(128, 4)         # Final output (4 classes)
        )

    def forward(self, sentence_input_ids, sentence_attention_mask, aspect_input_ids, aspect_attention_mask):
        # Get sentence embeddings
        sentence_outputs = self.bert(
            input_ids=sentence_input_ids,
            attention_mask=sentence_attention_mask
        )
        sentence_embedding = sentence_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Get aspect embeddings
        aspect_outputs = self.bert(
            input_ids=aspect_input_ids,
            attention_mask=aspect_attention_mask
        )
        aspect_embedding = aspect_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Combine features
        combined = torch.cat([sentence_embedding, aspect_embedding], dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)

        return logits
```

```
# Training Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model2 = AspectSentimentClassifier().to(device)
optimizer = AdamW(model2.parameters(), lr=2e-5)
```

```
class_counts = torch.tensor([
    91,    # conflict (class 0)
    805,   # negative (class 1)
    633,   # neutral (class 2)
    2164   # positive (class 3)
])

# Calculate weights (inverse frequency)
weights = 1. / class_counts.float()

# Normalize to sum to num_classes (4)
weights = weights / weights.sum() * len(class_counts)

# Verify weight-class alignment
```

```python
print({
    'conflict': weights[0].item(),  # Should be LARGEST (~3.75)
    'negative': weights[1].item(),   # ~1.23
    'neutral': weights[2].item(),    # ~1.57
    'positive': weights[3].item()    # Should be SMALLEST (~0.45)
})

# Initialize loss function
criterion = torch.nn.CrossEntropyLoss(weight=weights.to(device))
```

```
{'conflict': 3.0796351432800293, 'negative': 0.3481326401233673, 'neutral': 0.4427279531955719, 'positive': 0.12950406968593597}
```

```python
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
test_loader = DataLoader(test_dataset, batch_size=16)
```

Training

```python
# Training Loop
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss = 0

    for batch in dataloader:
        optimizer.zero_grad()

        inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
        labels = batch['labels'].to(device)

        outputs = model(**inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```python
def evaluate(model, dataloader, criterion, debug=False):
    model.eval()
    total_loss = 0.0
    predictions, true_labels = [], []
    class_losses = [0.0] * 4
    class_counts = [0] * 4

    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
            labels = batch['labels'].to(device)

            # Forward pass
            logits = model(**inputs)
            loss = criterion(logits, labels)
            total_loss += loss.item()

            # Per-class loss calculation
            for class_idx in range(4):
                mask = (labels == class_idx)
                if mask.any():
                    class_loss = criterion(logits[mask], labels[mask]).item()
                    class_losses[class_idx] += class_loss * mask.sum().item()
                    class_counts[class_idx] += mask.sum().item()

                    if debug:
                        print(f"Batch {batch_idx} Class {class_idx} Loss: {class_loss:.4f} (n={mask.sum().item()})")

            # Store predictions
            _, preds = torch.max(logits, dim=1)
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    # Calculate metrics
```

```python
    avg_loss = total_loss / len(dataloader)
    class_avg_losses = [
        class_losses[i] / class_counts[i] if class_counts[i] > 0 else 0.0
        for i in range(4)
    ]

    return {
        'loss': avg_loss,
        'class_losses': class_avg_losses,
        'class_counts': class_counts,
        'accuracy': accuracy_score(true_labels, predictions),
        'f1': f1_score(true_labels, predictions, average='macro')
    }
```

```python
best_val_loss = float('inf')  # Initialize with a very high value if tracking loss

# Training Loop
for epoch in range(10):
    model2.train()
    train_loss = train_epoch(model2, train_loader, optimizer, criterion)

    model2.eval()
    val_results = evaluate(model2, val_loader, criterion, debug=(epoch == 0))

    # Improved printing
    print(f"\nEpoch {epoch + 1}")
    print(f"Train Loss: {train_loss:.4f}")
    print(f"Val Loss: {val_results['loss']:.4f}")
    print(f"Val Accuracy: {val_results['accuracy']:.4f}")
    print(f"Val F1: {val_results['f1']:.4f}")

    # Class-wise performance
    class_names = ['conflict', 'negative', 'neutral', 'positive']
    print("\nClass-wise Val Loss:")
    for i, name in enumerate(class_names):
        if val_results['class_counts'][i] > 0:
            print(f"{name.upper():<9}: {val_results['class_losses'][i]:.4f} (n={val_results['class_counts'][i]})")
    print("-----")

    if val_results['loss'] < best_val_loss:
      best_val_loss = val_results['loss']
      torch.save({
          'epoch': epoch,
          'model_state_dict': model.state_dict(),
          'optimizer_state_dict': optimizer.state_dict(),
          'loss': train_loss,
          'val_loss': val_results['loss'],
          'val_accuracy': val_results['accuracy'],
      }, 'best_model2.pth')
      print(f"Saved new best model with val loss: {best_val_loss:.4f}")
```

```
NEUTRAL  : 1.4754 (n=173)
POSITIVE : 0.5450 (n=475)
-----

Epoch 9
Train Loss: 0.1812
Val Loss: 1.4461
Val Accuracy: 0.7313
Val F1: 0.5662

Class-wise Val Loss:
CONFLICT : 3.6173 (n=15)
NEGATIVE : 1.2714 (n=245)
NEUTRAL  : 1.4797 (n=173)
POSITIVE : 0.6819 (n=475)
-----

Epoch 10
Train Loss: 0.1636
Val Loss: 1.4153
Val Accuracy: 0.7390
Val F1: 0.5915

Class-wise Val Loss:
CONFLICT : 3.7869 (n=15)
NEGATIVE : 1.1181 (n=245)
NEUTRAL  : 1.4773 (n=173)
POSITIVE : 0.7178 (n=475)
-----
```

```python
checkpoint = torch.load('best_model2.pth')
print(f"Model 2 Best validation loss: {checkpoint['val_loss']:.4f}")
print(f"Achieved at epoch: {checkpoint['epoch'] + 1}")

# Evaluate on the test set ONLY ONCE
model2.eval()
test_results = evaluate(model2, test_loader, criterion)
print(f"\nModel 2 Final Test Performance:")
print(f"Test Loss: {test_results['loss']:.4f}")
print(f"Test Accuracy: {test_results['accuracy']:.4f}")
print(f"Test F1: {test_results['f1']:.4f}")
```

```
Model 2 Best validation loss: 0.7821
Achieved at epoch: 2

Model 2 Final Test Performance:
Test Loss: 1.5158
Test Accuracy: 0.7225
Test F1: 0.5759
```

Model 3

Model Architecture

```python
class AspectSentimentClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        self.dropout = torch.nn.Dropout(0.5)  # Increased from 0.2
        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(768*2, 512),
            torch.nn.ReLU(),
            torch.nn.LayerNorm(512),
            torch.nn.Dropout(0.3),  # Additional dropout
            torch.nn.Linear(512, 128),
            torch.nn.GELU(),
            torch.nn.Dropout(0.3),  # Additional dropout
            torch.nn.Linear(128, 4)
        )

    def forward(self, sentence_input_ids, sentence_attention_mask, aspect_input_ids, aspect_attention_mask):
        # Get sentence embeddings
        sentence_outputs = self.bert(
            input_ids=sentence_input_ids,
            attention_mask=sentence_attention_mask
        )
        sentence_embedding = sentence_outputs.last_hidden_state[:, 0, :]  # CLS token
```

```
        # Get aspect embeddings
        aspect_outputs = self.bert(
            input_ids=aspect_input_ids,
            attention_mask=aspect_attention_mask
        )
        aspect_embedding = aspect_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Combine features
        combined = torch.cat([sentence_embedding, aspect_embedding], dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)

        return logits
```

```
# Training Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model3 = AspectSentimentClassifier().to(device)
optimizer = AdamW(model3.parameters(), lr=2e-5)
```

```
class_counts = torch.tensor([
    91,    # conflict (class 0)
    805,   # negative (class 1)
    633,   # neutral (class 2)
    2164   # positive (class 3)
])

# Calculate weights (inverse frequency)
weights = 1. / class_counts.float()

# Normalize to sum to num_classes (4)
weights = weights / weights.sum() * len(class_counts)

# Verify weight-class alignment
print({
    'conflict': weights[0].item(),  # Should be LARGEST (~3.75)
    'negative': weights[1].item(),   # ~1.23
    'neutral': weights[2].item(),    # ~1.57
    'positive': weights[3].item()    # Should be SMALLEST (~0.45)
})

# Initialize loss function
criterion = torch.nn.CrossEntropyLoss(weight=weights.to(device))
```

```
{'conflict': 3.0796351432800293, 'negative': 0.3481326401233673, 'neutral': 0.4427279531955719, 'positive': 0.12950406968593597}
```

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
test_loader = DataLoader(test_dataset, batch_size=16)
```

Training

```
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss = 0

    for batch in dataloader:
        optimizer.zero_grad()

        inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
        labels = batch['labels'].to(device)

        outputs = model(**inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
def evaluate(model, dataloader, criterion, debug=False):
    model.eval()
```

```python
    total_loss = 0.0
    predictions, true_labels = [], []
    class_losses = [0.0] * 4
    class_counts = [0] * 4

    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
            labels = batch['labels'].to(device)

            # Forward pass
            logits = model(**inputs)
            loss = criterion(logits, labels)
            total_loss += loss.item()

            # Per-class loss calculation
            for class_idx in range(4):
                mask = (labels == class_idx)
                if mask.any():
                    class_loss = criterion(logits[mask], labels[mask]).item()
                    class_losses[class_idx] += class_loss * mask.sum().item()
                    class_counts[class_idx] += mask.sum().item()

                    if debug:
                        print(f"Batch {batch_idx} Class {class_idx} Loss: {class_loss:.4f} (n={mask.sum().item()})")

            # Store predictions
            _, preds = torch.max(logits, dim=1)
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    # Calculate metrics
    avg_loss = total_loss / len(dataloader)
    class_avg_losses = [
        class_losses[i] / class_counts[i] if class_counts[i] > 0 else 0.0
        for i in range(4)
    ]

    return {
        'loss': avg_loss,
        'class_losses': class_avg_losses,
        'class_counts': class_counts,
        'accuracy': accuracy_score(true_labels, predictions),
        'f1': f1_score(true_labels, predictions, average='macro')
    }
```

```python
best_val_loss = float('inf')  # Initialize with a very high value if tracking loss

# Training Loop
for epoch in range(10):
    model3.train()
    train_loss = train_epoch(model3, train_loader, optimizer, criterion)

    model3.eval()
    val_results = evaluate(model3, val_loader, criterion, debug=(epoch == 0))

    # Improved printing
    print(f"\nEpoch {epoch + 1}")
    print(f"Train Loss: {train_loss:.4f}")
    print(f"Val Loss: {val_results['loss']:.4f}")
    print(f"Val Accuracy: {val_results['accuracy']:.4f}")
    print(f"Val F1: {val_results['f1']:.4f}")

    # Class-wise performance
    class_names = ['conflict', 'negative', 'neutral', 'positive']
    print("\nClass-wise Val Loss:")
    for i, name in enumerate(class_names):
        if val_results['class_counts'][i] > 0:
            print(f"{name.upper():<9}: {val_results['class_losses'][i]:.4f} (n={val_results['class_counts'][i]})")
    print("-----")


    if val_results['loss'] < best_val_loss:
      best_val_loss = val_results['loss']
```

```
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': train_loss,
        'val_loss': val_results['loss'],
        'val_accuracy': val_results['accuracy'],
    }, 'best_model3.pth')
    print(f"Saved new best model with val loss: {best_val_loss:.4f}")
```

```
Batch 53 Class 2 Loss: 1.0365 (n=3)
Batch 53 Class 3 Loss: 0.8478 (n=11)
Batch 54 Class 1 Loss: 0.6652 (n=5)
Batch 54 Class 2 Loss: 0.5110 (n=2)
Batch 54 Class 3 Loss: 0.5096 (n=9)
Batch 55 Class 1 Loss: 1.8067 (n=5)
Batch 55 Class 2 Loss: 0.5187 (n=3)
Batch 55 Class 3 Loss: 0.7437 (n=8)
Batch 56 Class 0 Loss: 1.9382 (n=1)
Batch 56 Class 1 Loss: 1.0231 (n=2)
Batch 56 Class 2 Loss: 0.7201 (n=1)
Batch 56 Class 3 Loss: 0.6034 (n=8)

Epoch 1
Train Loss: 1.2560
Val Loss: 0.9752
Val Accuracy: 0.7269
Val F1: 0.5135

Class-wise Val Loss:
CONFLICT : 2.2395 (n=15)
NEGATIVE : 0.8106 (n=245)
NEUTRAL  : 0.9638 (n=173)
POSITIVE : 0.6212 (n=475)
-----
Saved new best model with val loss: 0.9752

Epoch 2
Train Loss: 0.9727
Val Loss: 0.8434
Val Accuracy: 0.6795
Val F1: 0.5668

Class-wise Val Loss:
CONFLICT : 0.4815 (n=15)
NEGATIVE : 0.8358 (n=245)
NEUTRAL  : 1.0082 (n=173)
POSITIVE : 0.8243 (n=475)
-----
Saved new best model with val loss: 0.8434

Epoch 3
Train Loss: 0.7939
Val Loss: 0.8444
Val Accuracy: 0.6949
Val F1: 0.5717

Class-wise Val Loss:
CONFLICT : 0.2897 (n=15)
NEGATIVE : 0.6274 (n=245)
NEUTRAL  : 1.2735 (n=173)
POSITIVE : 0.8722 (n=475)
-----

Epoch 4
Train Loss: 0.6402
Val Loss: 0.9843
Val Accuracy: 0.7048
Val F1: 0.5718
```

```
checkpoint = torch.load('best_model3.pth')
print(f"Model 3 Best validation loss: {checkpoint['val_loss']:.4f}")
print(f"Achieved at epoch: {checkpoint['epoch'] + 1}")

# Evaluate on the test set ONLY ONCE
model3.eval()
test_results = evaluate(model3, test_loader, criterion)
print(f"\nModel 3 Final Test Performance:")
print(f"Test Loss: {test_results['loss']:.4f}")
print(f"Test Accuracy: {test_results['accuracy']:.4f}")
print(f"Test F1: {test_results['f1']:.4f}")
```

```
Model 3 Best validation loss: 0.8434
Achieved at epoch: 2

Model 3 Final Test Performance:
Test Loss: 1.5514
```

```
Test Accuracy: 0.7313
Test F1: 0.5861
```

```
torch.save(model3.state_dict(), 'model3.pt')
```

Model 2: some enhancements

```
# Model Architecture
class AspectSentimentClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        self.dropout = torch.nn.Dropout(0.5)

        # Enhanced classifier with intermediate layers
        self.classifier = torch.nn.Sequential(
        torch.nn.Linear(768 * 2, 256),
        torch.nn.GELU(),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(256, 4)
    )

    def forward(self, sentence_input_ids, sentence_attention_mask, aspect_input_ids, aspect_attention_mask):
        # Get sentence embeddings
        sentence_outputs = self.bert(
            input_ids=sentence_input_ids,
            attention_mask=sentence_attention_mask
        )
        sentence_embedding = sentence_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Get aspect embeddings
        aspect_outputs = self.bert(
            input_ids=aspect_input_ids,
            attention_mask=aspect_attention_mask
        )
        aspect_embedding = aspect_outputs.last_hidden_state[:, 0, :]  # CLS token

        # Combine features
        combined = torch.cat([sentence_embedding, aspect_embedding], dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)

        return logits
```

```
# Training Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model4 = AspectSentimentClassifier().to(device)
optimizer = AdamW(model4.parameters(), lr=2e-5)
```

```
class_counts = torch.tensor([
    91,    # conflict (class 0)
    805,   # negative (class 1)
    633,   # neutral (class 2)
    2164   # positive (class 3)
])

# Calculate weights (inverse frequency)
weights = 1. / class_counts.float()

# Normalize to sum to num_classes (4)
weights = weights / weights.sum() * len(class_counts)

# Verify weight-class alignment
print({
    'conflict': weights[0].item(),  # Should be LARGEST (~3.75)
    'negative': weights[1].item(),   # ~1.23
    'neutral': weights[2].item(),    # ~1.57
    'positive': weights[3].item()    # Should be SMALLEST (~0.45)
})

# Initialize loss function
criterion = torch.nn.CrossEntropyLoss(weight=weights.to(device))
```

```
{'conflict': 3.0796351432800293, 'negative': 0.3481326401233673, 'neutral': 0.4427279531955719, 'positive': 0.12950406968593597}
```

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)
test_loader = DataLoader(test_dataset, batch_size=16)
```

```
# Training Loop
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss = 0

    for batch in dataloader:
        optimizer.zero_grad()

        inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
        labels = batch['labels'].to(device)

        outputs = model(**inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
def evaluate(model, dataloader, criterion, debug=False):
    model.eval()
    total_loss = 0.0
    predictions, true_labels = [], []
    class_losses = [0.0] * 4
    class_counts = [0] * 4

    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            inputs = {k: v.to(device) for k, v in batch.items() if k != 'labels'}
            labels = batch['labels'].to(device)

            # Forward pass
            logits = model(**inputs)
            loss = criterion(logits, labels)
            total_loss += loss.item()

            # Per-class loss calculation
            for class_idx in range(4):
                mask = (labels == class_idx)
                if mask.any():
                    class_loss = criterion(logits[mask], labels[mask]).item()
                    class_losses[class_idx] += class_loss * mask.sum().item()
                    class_counts[class_idx] += mask.sum().item()
```

```
                    if debug:
                        print(f"Batch {batch_idx} Class {class_idx} Loss: {class_loss:.4f} (n={mask.sum().item()})")

            # Store predictions
            _, preds = torch.max(logits, dim=1)
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    # Calculate metrics
    avg_loss = total_loss / len(dataloader)
    class_avg_losses = [
        class_losses[i] / class_counts[i] if class_counts[i] > 0 else 0.0
        for i in range(4)
    ]

    return {
        'loss': avg_loss,
        'class_losses': class_avg_losses,
        'class_counts': class_counts,
        'accuracy': accuracy_score(true_labels, predictions),
        'f1': f1_score(true_labels, predictions, average='macro')
    }
```

```
best_val_loss = float('inf')  # Initialize with a very high value if tracking loss

# Training Loop
for epoch in range(10):
    model4.train()
    train_loss = train_epoch(model4, train_loader, optimizer, criterion)

    model4.eval()
    val_results = evaluate(model4, val_loader, criterion, debug=(epoch == 0))

    # Improved printing
    print(f"\nEpoch {epoch + 1}")
    print(f"Train Loss: {train_loss:.4f}")
    print(f"Val Loss: {val_results['loss']:.4f}")
    print(f"Val Accuracy: {val_results['accuracy']:.4f}")
    print(f"Val F1: {val_results['f1']:.4f}")

    # Class-wise performance
    class_names = ['conflict', 'negative', 'neutral', 'positive']
    print("\nClass-wise Val Loss:")
    for i, name in enumerate(class_names):
        if val_results['class_counts'][i] > 0:
            print(f"{name.upper():<9}: {val_results['class_losses'][i]:.4f} (n={val_results['class_counts'][i]})")
    print("-----")

    if val_results['loss'] < best_val_loss:
      best_val_loss = val_results['loss']
      torch.save({
          'epoch': epoch,
          'model_state_dict': model.state_dict(),
          'optimizer_state_dict': optimizer.state_dict(),
          'loss': train_loss,
          'val_loss': val_results['loss'],
          'val_accuracy': val_results['accuracy'],
      }, 'best_model4.pth')
      print(f"Saved new best model with val loss: {best_val_loss:.4f}")
```

```
Batch 0 Class 1 Loss: 0.3849 (n=4)
Batch 0 Class 2 Loss: 1.0249 (n=6)
Batch 0 Class 3 Loss: 0.5856 (n=6)
Batch 1 Class 1 Loss: 0.6525 (n=4)
Batch 1 Class 2 Loss: 0.9147 (n=3)
Batch 1 Class 3 Loss: 0.7815 (n=9)
Batch 2 Class 0 Loss: 0.9841 (n=1)
Batch 2 Class 1 Loss: 1.1073 (n=3)
Batch 2 Class 2 Loss: 1.8111 (n=1)
Batch 2 Class 3 Loss: 0.5824 (n=11)
Batch 3 Class 1 Loss: 0.8340 (n=5)
Batch 3 Class 2 Loss: 0.9669 (n=3)
Batch 3 Class 3 Loss: 0.3203 (n=8)
Batch 4 Class 1 Loss: 0.9238 (n=6)
Batch 4 Class 2 Loss: 0.9084 (n=1)
```

```
Batch 4 Class 3 Loss: 0.4597 (n=9)
Batch 5 Class 0 Loss: 1.9898 (n=1)
Batch 5 Class 1 Loss: 1.5321 (n=3)
Batch 5 Class 2 Loss: 1.0775 (n=3)
Batch 5 Class 3 Loss: 1.0942 (n=9)
Batch 6 Class 1 Loss: 0.8515 (n=4)
Batch 6 Class 2 Loss: 1.0293 (n=5)
Batch 6 Class 3 Loss: 0.6266 (n=7)
Batch 7 Class 1 Loss: 0.8192 (n=7)
Batch 7 Class 2 Loss: 0.8901 (n=2)
Batch 7 Class 3 Loss: 0.8097 (n=7)
Batch 8 Class 1 Loss: 0.5659 (n=3)
Batch 8 Class 2 Loss: 1.1941 (n=4)
Batch 8 Class 3 Loss: 0.5593 (n=9)
Batch 9 Class 0 Loss: 1.2495 (n=1)
Batch 9 Class 1 Loss: 0.9731 (n=6)
Batch 9 Class 2 Loss: 1.2989 (n=3)
Batch 9 Class 3 Loss: 0.8743 (n=6)
Batch 10 Class 0 Loss: 1.0257 (n=1)
Batch 10 Class 1 Loss: 0.6681 (n=6)
Batch 10 Class 2 Loss: 1.3733 (n=3)
Batch 10 Class 3 Loss: 1.2136 (n=6)
Batch 11 Class 1 Loss: 1.2703 (n=3)
Batch 11 Class 2 Loss: 1.3025 (n=6)
Batch 11 Class 3 Loss: 0.6427 (n=7)
Batch 12 Class 1 Loss: 0.5157 (n=3)
Batch 12 Class 2 Loss: 1.2419 (n=5)
Batch 12 Class 3 Loss: 0.8705 (n=8)
Batch 13 Class 1 Loss: 0.7826 (n=2)
Batch 13 Class 2 Loss: 1.5956 (n=4)
Batch 13 Class 3 Loss: 0.6406 (n=10)
Batch 14 Class 1 Loss: 0.9521 (n=6)
Batch 14 Class 2 Loss: 1.8016 (n=1)
Batch 14 Class 3 Loss: 0.7448 (n=9)
Batch 15 Class 0 Loss: 1.2121 (n=1)
Batch 15 Class 1 Loss: 0.5191 (n=2)
Batch 15 Class 2 Loss: 1.1791 (n=4)
Batch 15 Class 3 Loss: 0.6061 (n=9)
Batch 16 Class 1 Loss: 0.4786 (n=2)
Batch 16 Class 2 Loss: 0.6926 (n=4)
Batch 16 Class 3 Loss: 0.9252 (n=10)
Batch 17 Class 0 Loss: 1.1472 (n=1)
```

```
checkpoint = torch.load('best_model4.pth')
print(f"Model 4 Best validation loss: {checkpoint['val_loss']:.4f}")
print(f"Achieved at epoch: {checkpoint['epoch'] + 1}")

# Evaluate on the test set ONLY ONCE
model4.eval()
test_results = evaluate(model4, test_loader, criterion)
print(f"\nModel 4 Final Test Performance:")
print(f"Test Loss: {test_results['loss']:.4f}")
print(f"Test Accuracy: {test_results['accuracy']:.4f}")
print(f"Test F1: {test_results['f1']:.4f}")
```

```
Model 4 Best validation loss: 0.8168
Achieved at epoch: 3

Model 4 Final Test Performance:
Test Loss: 1.3695
Test Accuracy: 0.7225
Test F1: 0.5784
```

```
# saving the model as model 4
torch.save(model4.state_dict(), 'model4.pt')
```

Using model

```
# Prepare inputs (example)
sentence = "The food was great but service was terrible"
aspect = "service"
```