STT AI Lab Assignment 7

Team 7

1. Soham Gaonkar 23110314
2. Chaitanya Sharma 23110072

Github Repo link : https://github.com/Soham-Gaonkar/CS203_Lab_07

Dataset preparation :

Dataset 1

```python
import pandas as pd

train_data_df = pd.read_csv('data/train.tsv', sep='\t', header=None, names=['text','label'])
train_data_df['label'] = train_data_df['label'].astype(int)
display(train_data_df.head())
# 0 is negative, 1 is positive
```

```python
from sklearn.model_selection import train_test_split

train_df, val_df = train_test_split(train_data_df, test_size=0.2, random_state=42)
print(train_df.shape, val_df.shape)
```

```
(5536, 2) (1384, 2)
```

Dataset 2

```python
import pandas as pd

imdb_df = pd.read_csv('data/imdb.csv')
imdb_df = imdb_df.rename(columns={'review':'text', 'sentiment':'label'})
```

```python
imdb_df['label'] = imdb_df['label'].apply(lambda x: 1 if x=='positive' else 0)
```

```python
from sklearn.model_selection import train_test_split

# split imdb data into train val and split and save
train_imdb_df, test_imdb_df = train_test_split(imdb_df, test_size=0.2, random_state=42)
train_imdb_df , val_imdb_df = train_test_split(train_imdb_df, test_size=0.2, random_state=42)

print(train_imdb_df.shape, val_imdb_df.shape, test_imdb_df.shape)

train_imdb_df.to_csv("data/train_imdb_split.csv", index=False)
val_imdb_df.to_csv("data/val_imdb_split.csv", index=False)
test_imdb_df.to_csv("data/test_imdb_split.csv", index=False)
```

```
(32000, 2) (8000, 2) (10000, 2)
```

Model Architecture:

```
MLP_Model(
  (layers): ModuleDict(
    (Layer 1): Sequential(
      (0): Linear(in_features=10000, out_features=512, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.3, inplace=False)
    )
    (Layer 2): Sequential(
      (0): Linear(in_features=512, out_features=256, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.3, inplace=False)
    )
    (Layer 3): Sequential(
      (0): Linear(in_features=256, out_features=128, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.3, inplace=False)
    )
    (Layer 4): Sequential(
      (0): Linear(in_features=128, out_features=64, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.3, inplace=False)
    )
    (Layer 5): Linear(in_features=64, out_features=2, bias=True)
  )
)
```

Total Params:

```
    sum_params = 0

    for name, param in model.named_parameters():
        print(f'Layer: {name} | Size: {param.size()} | Params: {param.numel()} | Trainable: {param.requires_grad}')
        if param.requires_grad:
            sum_params += param.numel()

    print(f'Total Trainable Parameters: {sum_params}')
```

```
Layer: layers.Layer 1.0.weight | Size: torch.Size([512, 10000]) | Params: 5120000 | Trainable: True
Layer: layers.Layer 1.0.bias | Size: torch.Size([512]) | Params: 512 | Trainable: True
Layer: layers.Layer 2.0.weight | Size: torch.Size([256, 512]) | Params: 131072 | Trainable: True
Layer: layers.Layer 2.0.bias | Size: torch.Size([256]) | Params: 256 | Trainable: True
Layer: layers.Layer 3.0.weight | Size: torch.Size([128, 256]) | Params: 32768 | Trainable: True
Layer: layers.Layer 3.0.bias | Size: torch.Size([128]) | Params: 128 | Trainable: True
Layer: layers.Layer 4.0.weight | Size: torch.Size([64, 128]) | Params: 8192 | Trainable: True
Layer: layers.Layer 4.0.bias | Size: torch.Size([64]) | Params: 64 | Trainable: True
Layer: layers.Layer 5.weight | Size: torch.Size([2, 64]) | Params: 128 | Trainable: True
Layer: layers.Layer 5.bias | Size: torch.Size([2]) | Params: 2 | Trainable: True
Total Trainable Parameters: 5293122
```

Bag of words:

```
    BATCH_SIZE = 32

    # fit CountVectorizer on training data
    vectorizer = CountVectorizer(max_features=10000)
    vectorizer.fit(train_df["text"])

    def bow_collate_fn(batch):
        texts, labels = zip(*batch)  # Unpack batch into texts and labels
        X_batch = vectorizer.transform(texts).toarray()  # Vectorize texts using BoW
        X_batch_tensor = torch.tensor(X_batch, dtype=torch.float32)  # Convert to tensor
        y_batch_tensor = torch.tensor(labels, dtype=torch.long)  # Convert labels to tensor
        return X_batch_tensor, y_batch_tensor

    train_loader_bow = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=bow_collate_fn)
    val_loader_bow = DataLoader(val_dataset, batch_size=BATCH_SIZE,shuffle = False, collate_fn=bow_collate_fn)
    test_loader_bow = DataLoader(test_dataset, batch_size=BATCH_SIZE,shuffle= False, collate_fn=bow_collate_fn)

    print(f"Number of training batches: {len(train_loader_bow)}")
    print(f"Number of validation batches: {len(val_loader_bow)}")
    print(f"Number of test batches: {len(test_loader_bow)}")
```

```
Number of training batches: 173
Number of validation batches: 44
Number of test batches: 57
```

Bert Embedding :

```python
from transformers import AutoTokenizer, AutoModel

model_name = 'bert-base-uncased'

tokenizer = AutoTokenizer.from_pretrained(model_name)
print(tokenizer)

bert = AutoModel.from_pretrained(model_name)

embedding_size = bert.config.hidden_size
print('Embedding size:',embedding_size)
```

```
BertTokenizerFast(name_or_path='bert-base-uncased', vocab_size=30522, model_max_length=512, is_fast=True, padding
        0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
}
)
Embedding size: 768
```

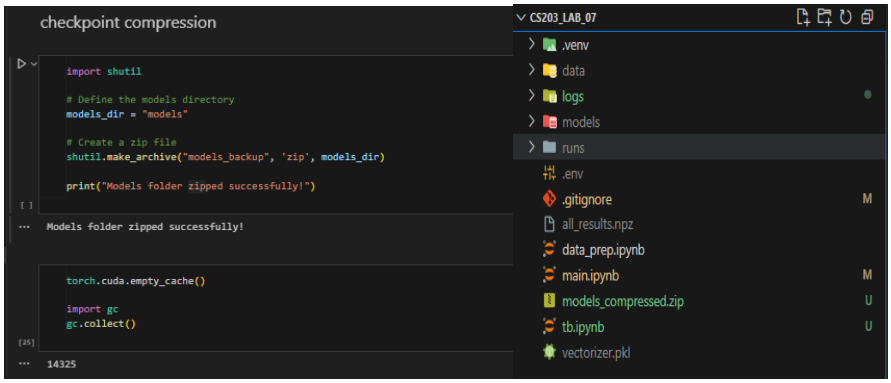
```python
def get_bert_embeddings(inputs, bert, pooling_strategy='mean'):
    with torch.no_grad():
        outputs = bert(**inputs)

    embeddings = outputs.last_hidden_state

    if pooling_strategy == 'mean':
        sentence_embeddings = torch.mean(embeddings, dim=1)  # Mean pooling
    elif pooling_strategy == 'max':
        sentence_embeddings = torch.max(embeddings, dim=1).values  # Max pooling
    elif pooling_strategy == 'cls':
        sentence_embeddings = embeddings[:, 0, :]  # CLS token
    elif pooling_strategy == 'sep':
        sentence_embeddings = embeddings[:, -1, :]  # SEP token
    else:
        raise ValueError("Pooling strategy not valid")

    return sentence_embeddings

def bert_collate_fn(batch):
    texts = [item[0] for item in batch]
    labels = None
    if isinstance(batch[0], tuple) and len(batch[0]) > 1:
        labels = torch.tensor([item[1] for item in batch], dtype=torch.long)

    inputs = tokenizer(texts, return_tensors="pt", padding=True, truncation=True).to(device)
    sentence_embeddings = get_bert_embeddings(inputs, bert, pooling_strategy=pooling_strategy)

    return sentence_embeddings.cpu() , labels

BATCH_SIZE = 32
pooling_strategy = 'mean'

train_loader_bert = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=bert_collate_fn)
val_loader_bert = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=bert_collate_fn)
test_loader_bert = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=bert_collate_fn)

print(f"Number of training batches: {len(train_loader_bert)}")
print(f"Number of validation batches: {len(val_loader_bert)}")
print(f"Number of test batches: {len(test_loader_bert)}")
```

```
Number of training batches: 173
Number of validation batches: 44
Number of test batches: 57
```

Train:

```python
import torch
import torch.nn as nn

def train(model, model_path, optimizer, criterion, train_loader, val_loader, num_epochs, device):
    model.to(device)
    best_loss = float('inf')
    train_loss_history = []
    val_loss_history = []
    best_model_state = None

    best_acc =0

    for epoch in range(1, num_epochs + 1):
        model.train()
        train_loss = 0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_loss /= len(train_loader)

        model.eval()
        val_loss = 0
        val_acc = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                X_batch, y_batch = X_batch.to(device), y_batch.to(device)

                y_pred_val = model(X_batch)
                loss = criterion(y_pred_val, y_batch)
                val_loss += loss.item()

                y_pred_val_class = torch.argmax(y_pred_val, dim=1)
                correct += torch.sum(y_pred_val_class == y_batch).item()
                total += y_batch.size(0)

        val_loss /= len(val_loader)
        val_acc = correct / total

        print(f'Epoch: {epoch} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.4f}')

        train_loss_history.append(train_loss)
        val_loss_history.append(val_loss)

        # Save checkpoint after every epoch
        checkpoint = {
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'train_loss': train_loss,
            'val_loss': val_loss,
            'val_acc': val_acc,
        }
        if epoch%2 ==0 :
            torch.save(checkpoint, f'{model_path}_epoch_{epoch}.pt')

        # if val_loss < best_loss:
        #     best_loss = val_loss
        #     best_model_state = model.state_dict()

        if val_acc > best_acc:
            best_acc = val_acc
            best_model_state = model.state_dict()

    print('Best val acc:', best_acc)
    torch.save(best_model_state, model_path)

    return train_loss_history, val_loss_history
```

Check point compression:



Training Hyperparams:

```python
bow_model = MLP_Model(input_size=10000)
bow_model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(bow_model.parameters(), lr=1e-4)

num_epochs = 10

epochs = list(range(1, num_epochs + 1))
train_loss_history, val_loss_history = train(
    bow_model,
    'models/bow_1.pt',
    optimizer,
    criterion,
    train_loader_bow,
    val_loader_bow,
    num_epochs,
    device
)
```

```python
bert_model = MLP_Model(input_size=embedding_size)
bert_model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(bert_model.parameters(), lr=1e-4)

num_epochs = 10

epochs = list(range(1, num_epochs + 1))
train_loss_history3, val_loss_history3 = train(
    bert_model,
    'models/bert_1.pt',
    optimizer,
    criterion,
    train_loader_bert,
    val_loader_bert,
    num_epochs,
    device
)
```

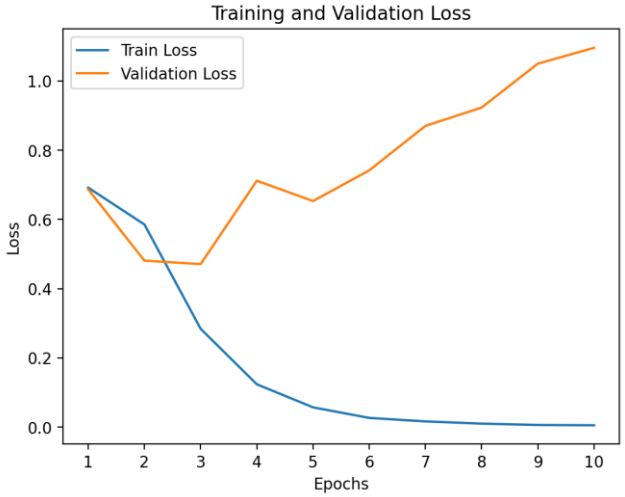
```python
# Load the check pointed model
bow_model_2 = MLP_Model(input_size=10000)
bow_model_2.load_state_dict(torch.load('models/bow_1.pt'))
bow_model_2.to(device)

print('model loaded')

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(bow_model_2.parameters(), lr=1e-4)

num_epochs = 10

epochs = list(range(1, num_epochs + 1))
train_loss_history, val_loss_history = train(
    bow_model_2,
    'models/bow_2.pt',
    optimizer,
    criterion,
    train_loader_bow_imdb,
    val_loader_bow_imdb,
    num_epochs,
    device
)
```

```python
# Load the check pointed model
bert_model_2 = MLP_Model(input_size=embedding_size)
bert_model_2.load_state_dict(torch.load('models/bert_1.pt'))
bert_model_2.to(device)

print('model loaded')

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(bert_model_2.parameters(), lr=1e-4)

num_epochs = 10

epochs = list(range(1, num_epochs + 1))
train_loss_history, val_loss_history = train(
    bert_model_2,
    'models/bert_2.pt',
    optimizer,
    criterion,
    train_loader_bert_imdb,
    val_loader_bert_imdb,
    num_epochs,
    device
)
```

(batch size is taken as 32 everywhere , every training consists of 10 epochs only)

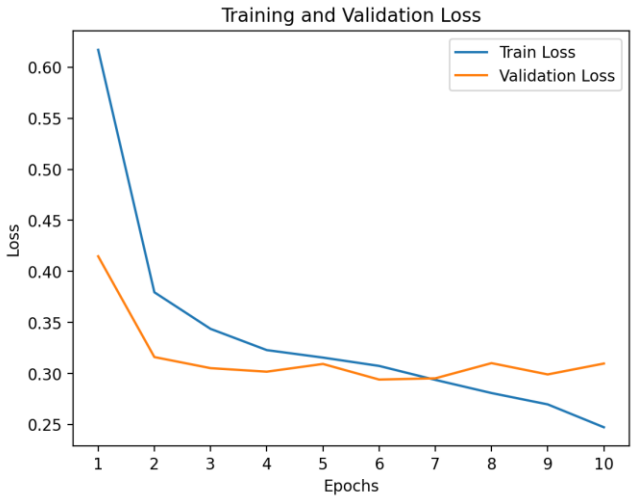
Training Bow Model on Dataset 1:

```
...   Epoch: 1 | Train Loss: 0.6917 | Val Loss: 0.6868 | Val Acc: 0.5181
      Epoch: 2 | Train Loss: 0.5854 | Val Loss: 0.4810 | Val Acc: 0.7760
      Epoch: 3 | Train Loss: 0.2842 | Val Loss: 0.4710 | Val Acc: 0.8027
      Epoch: 4 | Train Loss: 0.1238 | Val Loss: 0.7117 | Val Acc: 0.7616
      Epoch: 5 | Train Loss: 0.0573 | Val Loss: 0.6531 | Val Acc: 0.8078
      Epoch: 6 | Train Loss: 0.0267 | Val Loss: 0.7416 | Val Acc: 0.8100
      Epoch: 7 | Train Loss: 0.0166 | Val Loss: 0.8702 | Val Acc: 0.7926
      Epoch: 8 | Train Loss: 0.0102 | Val Loss: 0.9232 | Val Acc: 0.7941
      Epoch: 9 | Train Loss: 0.0063 | Val Loss: 1.0499 | Val Acc: 0.7934
      Epoch: 10 | Train Loss: 0.0054 | Val Loss: 1.0957 | Val Acc: 0.7948
      Best val acc: 0.809971098265896
```



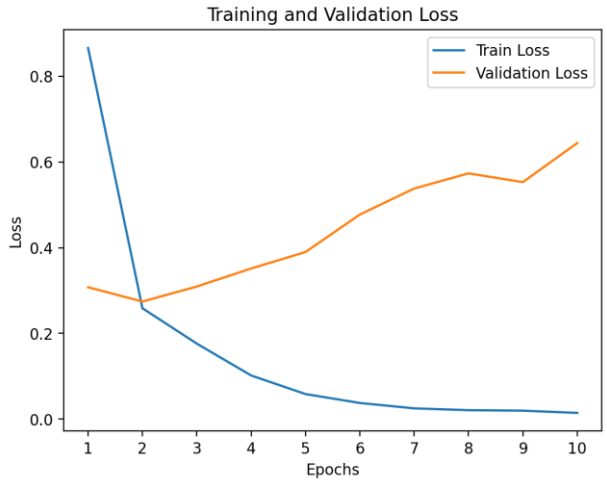
Training Bert model on Dataset 1:

```
Epoch: 1 | Train Loss: 0.6170 | Val Loss: 0.4148 | Val Acc: 0.8129
Epoch: 2 | Train Loss: 0.3796 | Val Loss: 0.3161 | Val Acc: 0.8540
Epoch: 3 | Train Loss: 0.3437 | Val Loss: 0.3052 | Val Acc: 0.8605
Epoch: 4 | Train Loss: 0.3229 | Val Loss: 0.3018 | Val Acc: 0.8736
Epoch: 5 | Train Loss: 0.3156 | Val Loss: 0.3093 | Val Acc: 0.8548
Epoch: 6 | Train Loss: 0.3074 | Val Loss: 0.2941 | Val Acc: 0.8678
Epoch: 7 | Train Loss: 0.2938 | Val Loss: 0.2953 | Val Acc: 0.8707
Epoch: 8 | Train Loss: 0.2808 | Val Loss: 0.3101 | Val Acc: 0.8605
Epoch: 9 | Train Loss: 0.2698 | Val Loss: 0.2991 | Val Acc: 0.8656
Epoch: 10 | Train Loss: 0.2473 | Val Loss: 0.3097 | Val Acc: 0.8714
Best val acc: 0.8735549132947977
```



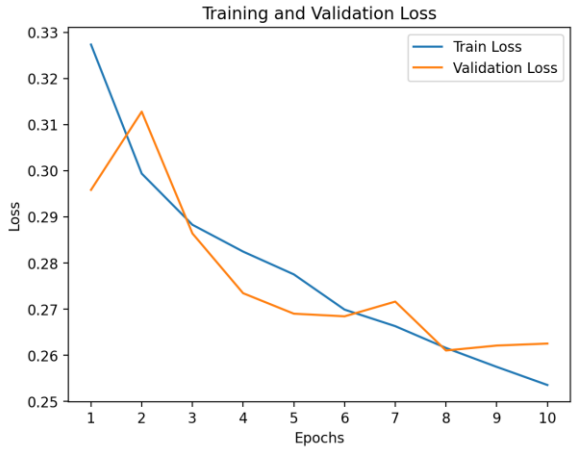
Tuning Bow on Dataset 2:

```
<ipython-input-27-eda9b00f0f99>:5: FutureWarning: You are using `torch.load
  bow_model_2.load_state_dict(torch.load('models/bow_1.pt'))
model loaded
Epoch: 1 | Train Loss: 0.8659 | Val Loss: 0.3077 | Val Acc: 0.8720
Epoch: 2 | Train Loss: 0.2586 | Val Loss: 0.2743 | Val Acc: 0.8892
Epoch: 3 | Train Loss: 0.1761 | Val Loss: 0.3093 | Val Acc: 0.8830
Epoch: 4 | Train Loss: 0.1018 | Val Loss: 0.3514 | Val Acc: 0.8838
Epoch: 5 | Train Loss: 0.0584 | Val Loss: 0.3898 | Val Acc: 0.8848
Epoch: 6 | Train Loss: 0.0376 | Val Loss: 0.4773 | Val Acc: 0.8850
Epoch: 7 | Train Loss: 0.0250 | Val Loss: 0.5380 | Val Acc: 0.8845
Epoch: 8 | Train Loss: 0.0208 | Val Loss: 0.5733 | Val Acc: 0.8838
Epoch: 9 | Train Loss: 0.0197 | Val Loss: 0.5530 | Val Acc: 0.8866
Epoch: 10 | Train Loss: 0.0144 | Val Loss: 0.6441 | Val Acc: 0.8822
Best val acc: 0.88925
```



Tuning Bert on Dataset 2:

```
<ipython-input-29-884dbd6df4d1>:5: FutureWarning: You are using `torch.load`
  bert_model_2.load_state_dict(torch.load('models/bert_1.pt'))
model loaded
Epoch: 1 | Train Loss: 0.3273 | Val Loss: 0.2958 | Val Acc: 0.8722
Epoch: 2 | Train Loss: 0.2994 | Val Loss: 0.3128 | Val Acc: 0.8645
Epoch: 3 | Train Loss: 0.2883 | Val Loss: 0.2864 | Val Acc: 0.8771
Epoch: 4 | Train Loss: 0.2825 | Val Loss: 0.2735 | Val Acc: 0.8842
Epoch: 5 | Train Loss: 0.2775 | Val Loss: 0.2690 | Val Acc: 0.8869
Epoch: 6 | Train Loss: 0.2699 | Val Loss: 0.2685 | Val Acc: 0.8895
Epoch: 7 | Train Loss: 0.2663 | Val Loss: 0.2716 | Val Acc: 0.8848
Epoch: 8 | Train Loss: 0.2616 | Val Loss: 0.2610 | Val Acc: 0.8912
Epoch: 9 | Train Loss: 0.2575 | Val Loss: 0.2621 | Val Acc: 0.8920
Epoch: 10 | Train Loss: 0.2536 | Val Loss: 0.2626 | Val Acc: 0.8919
Best val acc: 0.892
```



Testing :

```python
import torch
import torch.nn.functional as F
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns

import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

def test(model, criterion, test_loader, device):
    model.eval()
    test_loss = 0
    all_y_true = []
    all_y_pred = []

    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            # Forward pass
            y_pred_logits = model(X_batch)  # shape: (batch_size, num_classes)
            loss = criterion(y_pred_logits, y_batch)
            test_loss += loss.item()

            # Get predicted labels
            y_pred_labels = torch.argmax(y_pred_logits, dim=1)  # Predicted class indices

            # Collect true and predicted labels for metrics
            all_y_true.extend(y_batch.cpu().numpy())
            all_y_pred.extend(y_pred_labels.cpu().numpy())

    # Average test loss over all batches
    test_loss /= len(test_loader)

    # Compute metrics
    accuracy = accuracy_score(all_y_true, all_y_pred)
    conf_matrix = confusion_matrix(all_y_true, all_y_pred)

    print(f'Test Loss: {test_loss:.4f}')
    print(f'Accuracy: {accuracy:.4f}')

    # Plot confusion matrix
    plt.figure(figsize=(6, 4))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1])
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title("Confusion Matrix")
    plt.show()

    return accuracy, conf_matrix

accuracy_bow_1, conf_matrix_bow_1 = test(bow_model, criterion, test_loader_bow, device)
```

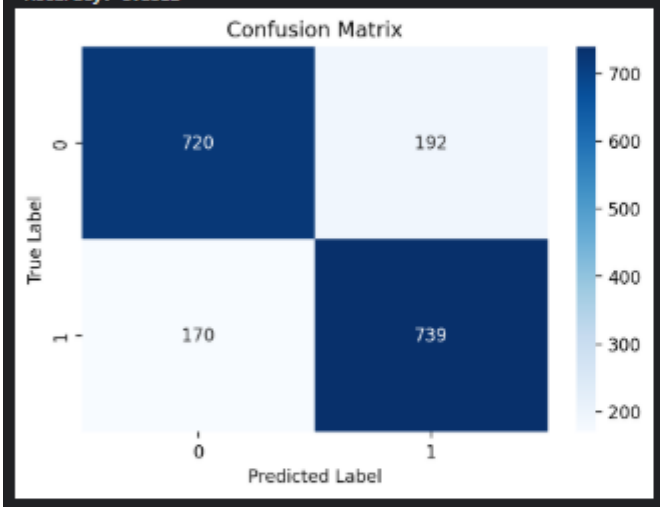
Results:

Model	Dataset 1 Accuracy	Dataset 2 Accuracy
bow_1	0.8012	0.7633
bow_2	0.7963	0.8826
bert_1	0.8655	0.8119
bert_2	0.8342	0.8890

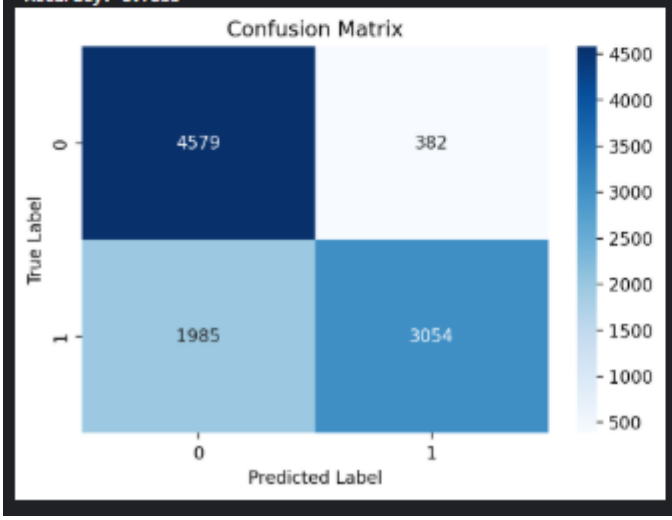
BOW trained on dataset 1

```
# #### BOW Trained on Dataset 1

# %%
bow_model = MLP_Model(input_size=10000, hidden_sizes=[512, 256, 128, 64], output_size=2, dropout_rate=0.3)
bow_model.load_state_dict(torch.load('/kaggle/input/models/bow_1.pt', map_location=torch.device('cpu')))
bow_model.to(device)  # Move model to device

criterion = nn.CrossEntropyLoss()

accuracy_bow_11, conf_matrix_bow_11 = test(bow_model, criterion, test_loader_bow, device)
accuracy_bow_12 , conf_matrix_bow_12 = test(bow_model, criterion, test_loader_bow_imdb, device)

# %% [markdown]
```

```
<ipython-input-4-adbab20603e8>:5: FutureWarning: You are using `torch.load` with `weights_only=False` (the current defaul
pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_onl
ly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True`
  bow_model.load_state_dict(torch.load('/kaggle/input/models/bow_1.pt', map_location=torch.device('cpu')))
Test Loss: 1.1444
Accuracy: 0.8012
```



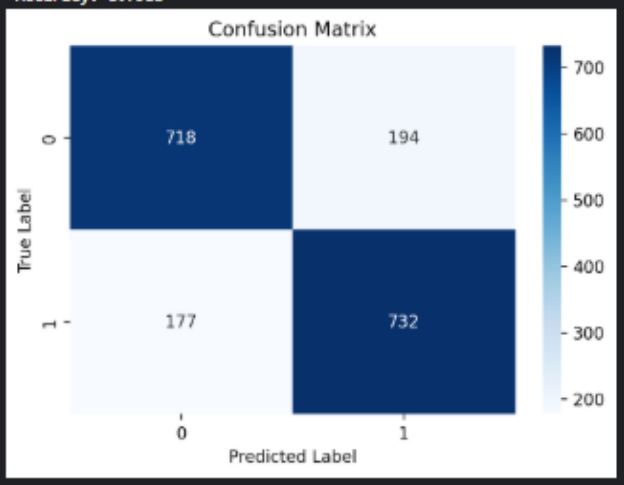

```
Test Loss: 7.9610
Accuracy: 0.7633
```



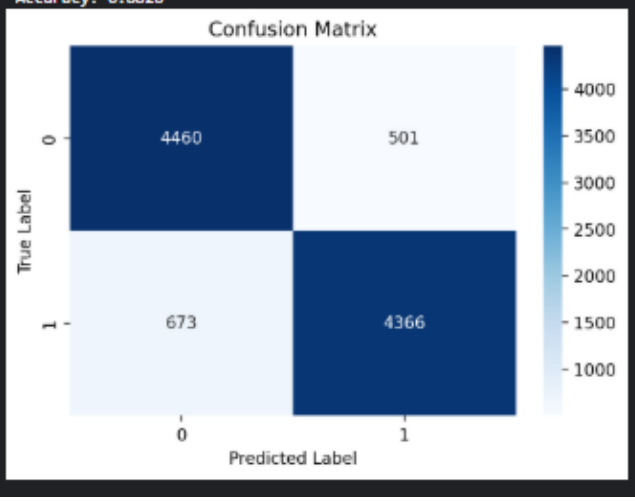
BOW Tuned on Dataset 2:

```
# #### Bow Tuned on Dataset 2

# %%

bow_model2 = MLP_Model(input_size=10000, hidden_sizes=[512, 256, 128, 64], output_size=2, dropout_rate=0.3)
bow_model2.load_state_dict(torch.load('/kaggle/input/models/bow_2.pt', map_location=torch.device('cpu')))
bow_model2.to(device)  # Move model to device

criterion = nn.CrossEntropyLoss()

accuracy_bow_21, conf_matrix_bow_21 = test(bow_model2, criterion, test_loader_bow, device)
accuracy_bow_22, conf_matrix_bow_22 = test(bow_model2, criterion, test_loader_bow_imdb, device)
```

```
<ipython-input-5-042af01c3d3a>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default va
pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` w
ly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for
  bow_model2.load_state_dict(torch.load('/kaggle/input/models/bow_2.pt', map_location=torch.device('cpu')))
Test Loss: 0.9520
Accuracy: 0.7963
```



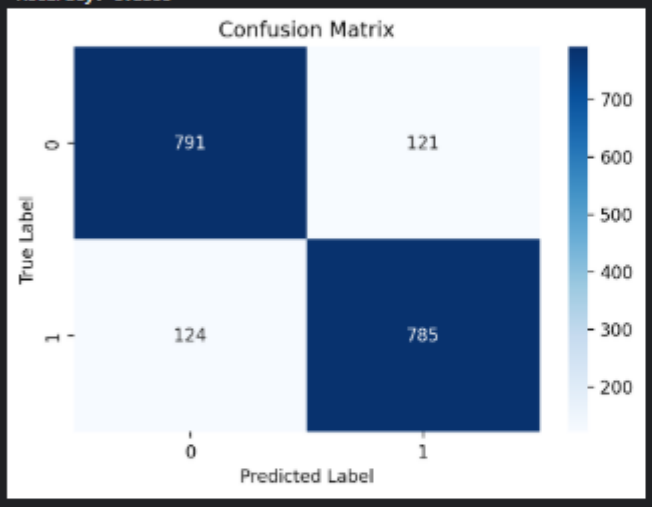

```
Test Loss: 0.6396
Accuracy: 0.8826
```



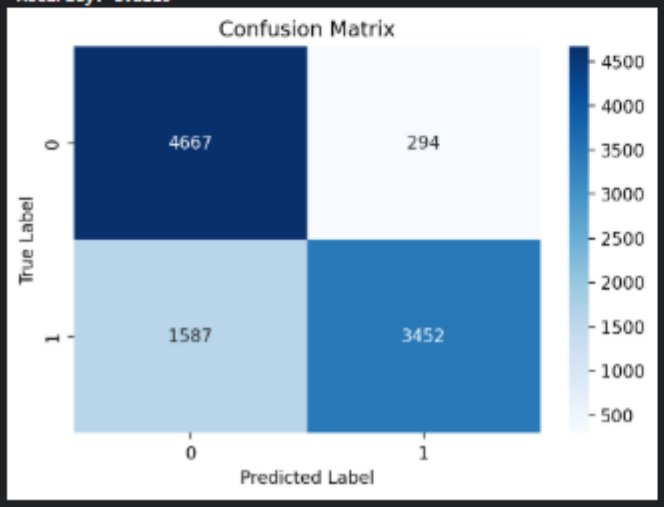
BERT Trained on Dataset 1:

```
# %% [markdown]
# #### Bert Trained on Dataset 1
# %%
bert_model = MLP_Model(input_size=768, hidden_sizes=[512, 256, 128, 64], output_size=2, dropout_rate=0.3)
bert_model.load_state_dict(torch.load('/kaggle/input/models/bert_1.pt', map_location=torch.device('cpu')))
bert_model.to(device)


criterion = nn.CrossEntropyLoss()

accuracy_bert_11, conf_matrix_bert_11 = test(bert_model, criterion, test_loader_bert, device)
accuracy_bert_12, conf_matrix_bert_12 = test(bert_model, criterion, test_loader_bert_imdb, device)
```

```
<ipython-input-6-e6c9928a5124>:5: FutureWarning: You are using `torch.load` with `weights_only=False` (the current defau
pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_on
ly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True
  bert_model.load_state_dict(torch.load('/kaggle/input/models/bert_1.pt', map_location=torch.device('cpu')))
Test Loss: 0.3167
Accuracy: 0.8655
```



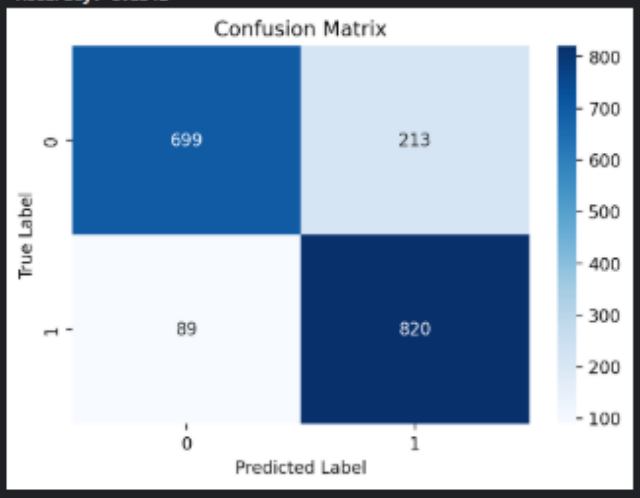

```
Test Loss: 0.4268
Accuracy: 0.8119
```



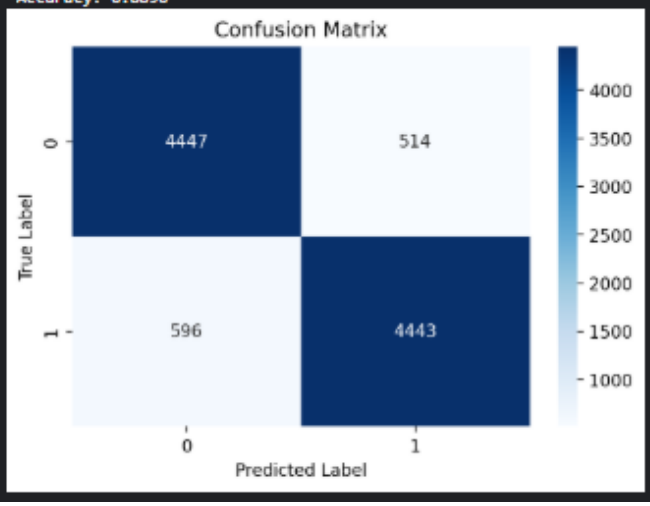
BERT Tuned on Dataset 2:

```
# %% [markdown]
# #### Bert Tuned on Dataset 2

# %%
bert_model_2 = MLP_Model(input_size=768, hidden_sizes=[512, 256, 128, 64], output_size=2, dropout_rate=0.3)
bert_model_2.load_state_dict(torch.load('/kaggle/input/models/bert_2.pt', map_location=torch.device('cpu')))
bert_model_2.to(device)

criterion = nn.CrossEntropyLoss()

accuracy_bow_21, conf_matrix_bow_21 = test(bert_model_2, criterion, test_loader_bert, device)
accuracy_bow_22, conf_matrix_bow_22 = test(bert_model_2, criterion, test_loader_bert_imdb, device)
```

```
<ipython-input-7-ccbf47d4945a>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default v
pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only`
ly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` fo
  bert_model_2.load_state_dict(torch.load('/kaggle/input/models/bert_2.pt', map_location=torch.device('cpu')))
Test Loss: 0.4257
Accuracy: 0.8342
```

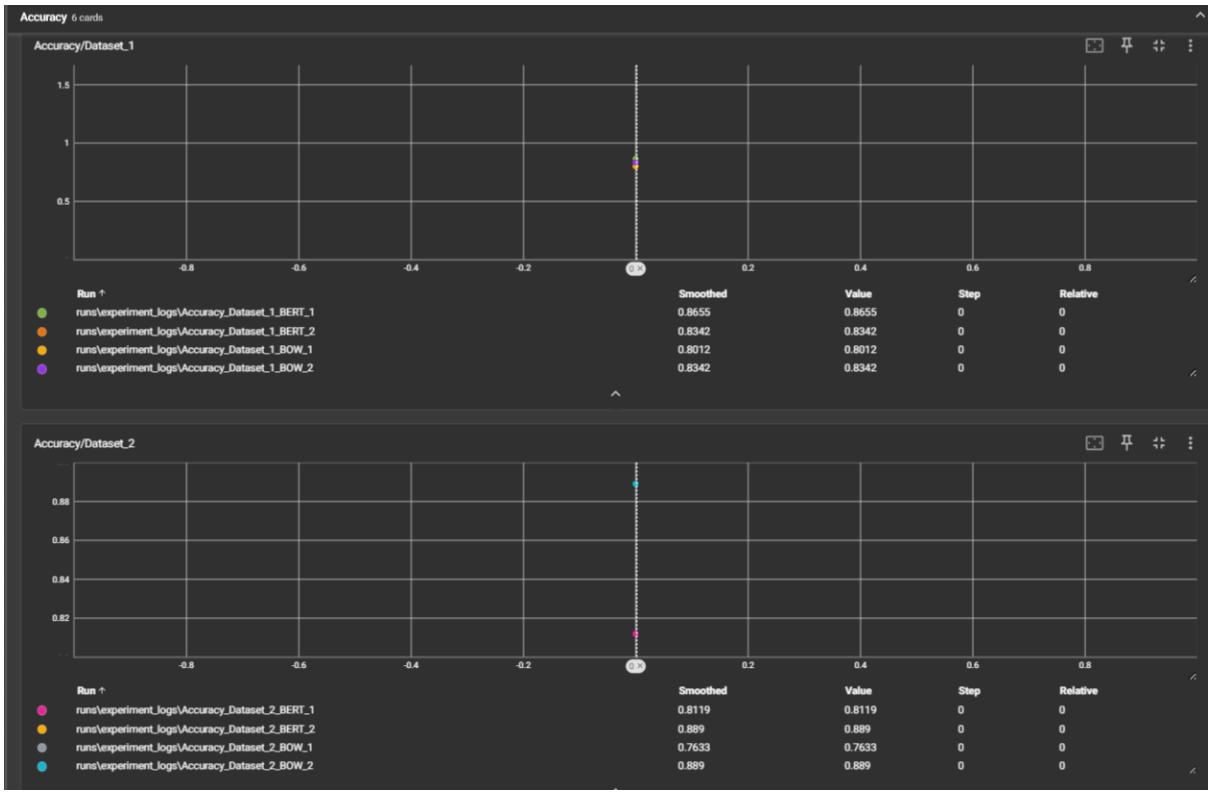



```
Test Loss: 0.2663
Accuracy: 0.8890
```



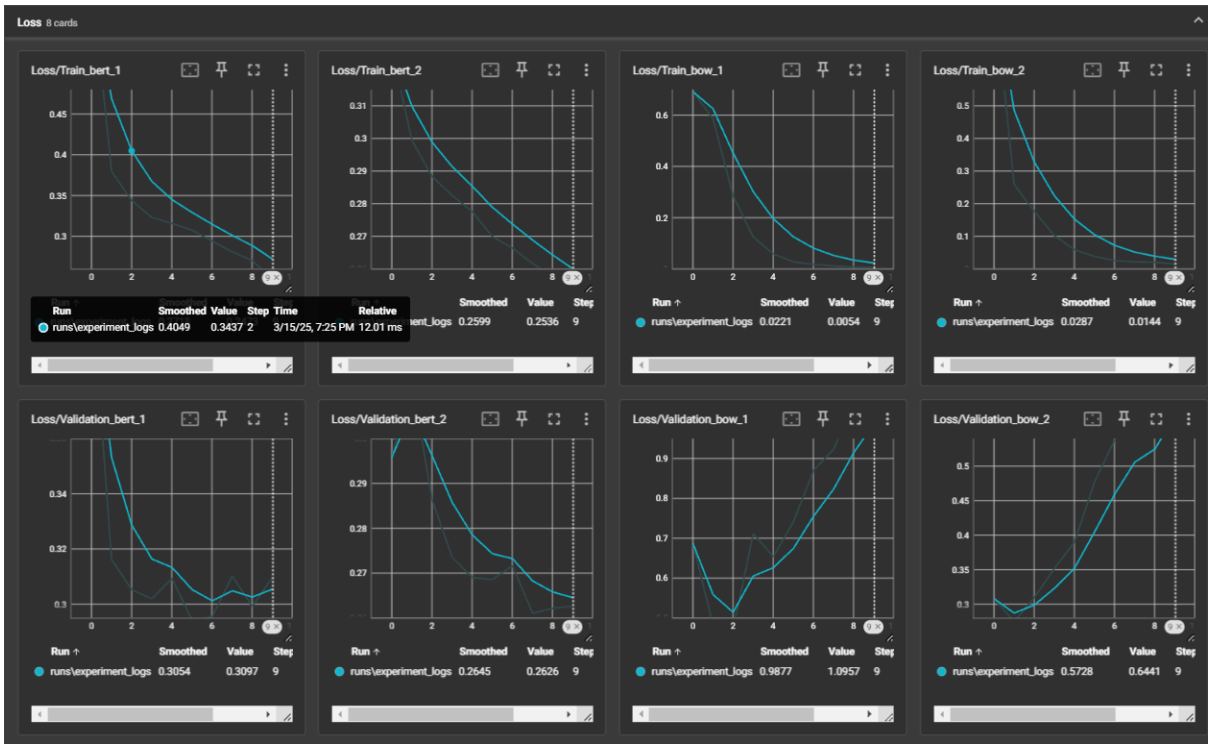
Tensorboard Integration :

Accuracy on Both datasets:



Validation Acc curves:

Validation Loss Curves:



Confusion Matrix :

Conclusion :

BoW Model Performance

- On **Dataset 1**, the original BoW model (bow_1) achieved **80.12%** accuracy.
- On **Dataset 2**, bow_1 performed at **76.33%**, but after fine-tuning (bow_2), it improved significantly to **88.26%**. This highlights that adapting the model to the IMDB dataset allowed it to generalize better to its specific language patterns, improving test accuracy.

BERT Embedding Model Performance

- On **Dataset 1**, the BERT-based model (bert_1) had a strong baseline performance of **86.55%**, outperforming BoW.
- On **Dataset 2**, bert_1 started with **81.19%** accuracy, but after fine-tuning (bert_2), it improved to **88.9%**, indicating that additional training on the IMDB dataset helped the model adapt better to its specific linguistic patterns and sentiment structures.

Fine-tuning significantly boosts a model's performance on a target dataset, as seen in bow_2 and bert_2 on Dataset 2. However, this often comes at the cost of slight performance degradation on the original dataset, highlighting a trade-off between generalization and dataset specialization.

- **BoW has more features but lacks context and sentence structure.** It relies on word frequency, making it limited in capturing deeper meaning.
- **BERT embeddings, despite having fewer dimensions, capture more meaningful relationships between words and sentences** due to their deep contextual understanding.

This explains why BERT-based models (bert_1 and bert_2) achieved **higher baseline accuracy than BoW**, as they leveraged better feature representations rather than just raw word counts.