MI201 – Project Report

Development of Text Emotion Classifiers using Machine Learning Techniques

Gabriel Dupuis, Louis Marchal

February 7, 2025

Contents

Introduction

In the digital age, Internet moderation has become crucial to ensure a safe and respectful communication space, helping to limit the spread of abusive or inappropriate content.

This project therefore aims to develop a sentiment classification system based on text embeddings, in the context of content moderation on Twitter.

By leveraging the contextual representations generated by pre-trained models such as BERT, we seek to efficiently identify the tone of different messages (positive, neutral, or negative).

The main challenge is to design a moderation bot capable of automatically detecting and flagging potentially problematic messages, relying on classical machine learning techniques as well as more complex architectures like multi-layer perceptron (MLP) neural networks or fine-tuning of language models (LLM).

Q0: Data Analysis using Classical Unsupervised Machine Learning Methods

In this section, we aim to understand the data we are working with and analyze it using various classical unsupervised machine learning methods (PCA, TSNE, KMeans).

Data Format

```
train_dataset = path+'/train.csv'
test_dataset = path+'/test.csv'

# Check if the path exists
print (os.path.exists(train_dataset))
print (os.path.exists(test_dataset))

# Load the CSV file into a DataFrame
train_df = pd.read_csv(train_dataset, encoding='ISO-8859-1')
test_df = pd.read_csv(test_dataset, encoding='ISO-8859-1')
```

First, we examine what the texts we will work on look like and their distribution by class. We have chosen texts from Twitter to work on real messages.

```
# Distribution of sentiment labels
sns.countplot(x='sentiment', data=train_df)
plt.title("Distribution of Sentiment Labels")
plt.show()
```

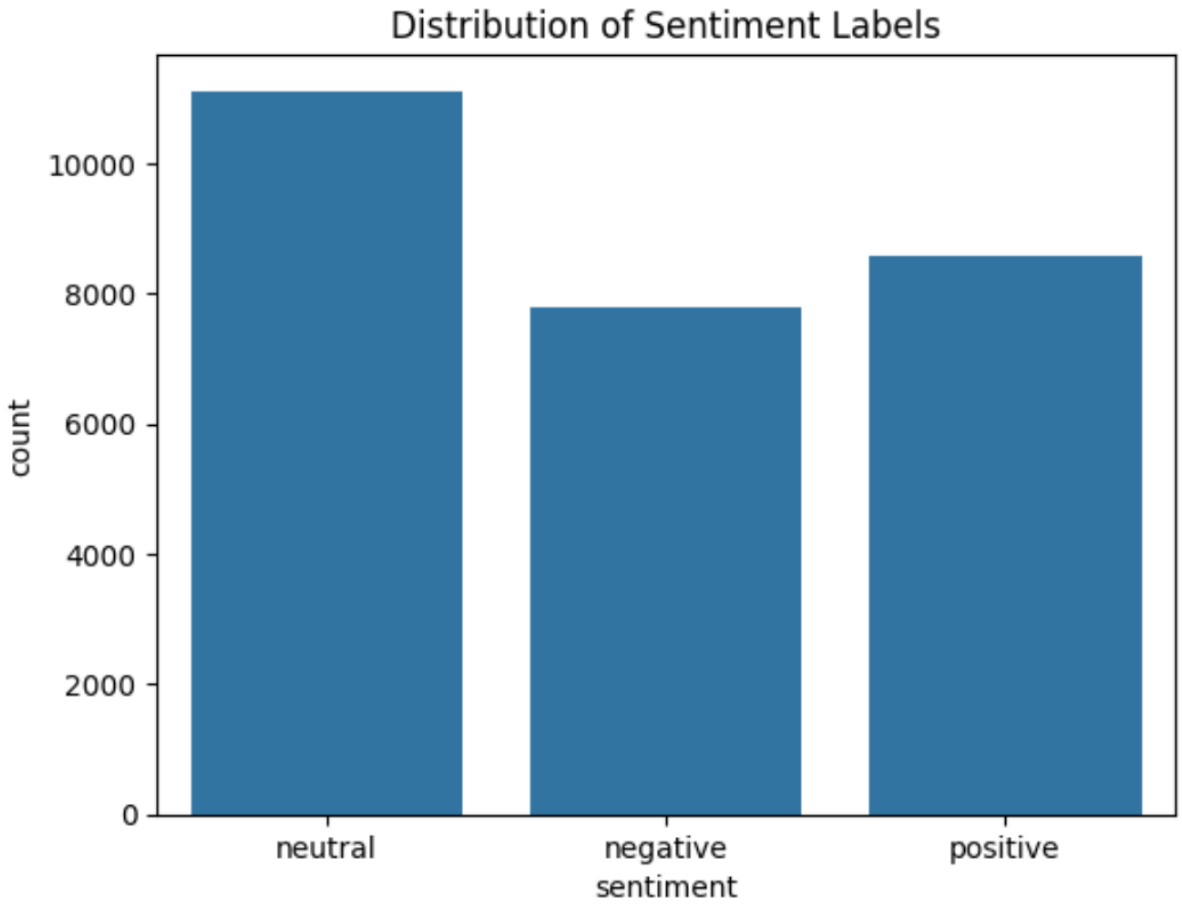

Figure 1: Histogram by Class

We apply preprocessing on the text to remove all stop-words, leaving only the words that are of interest.

Note: Since we are working with texts from messages that are not in perfect English, and which include emojis and special characters, this step is very important to isolate the truly important content and improve the quality of our predictions.

```
# Text preprocessing function that removes stopwords and converts text to lowercase

def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()
    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    text = " ".join([word for word in text.split() if word not in stop_words])
    return text
```

```python
# Apply preprocessing on train dataset
train_df['processed_text'] = train_df['text'].apply(preprocess_text)
train_df.head()
```

	textID	text	selected_text	sentiment	Time of Tweet	Age of User	Country	Population -2020	Land Area (Km²)	Density (P/Km²)	sentiment_class	processed_text
0	cb774db0d1	I`d have responded, if I were going	I`d have responded, if I were going	neutral	morning	0-20	Afghanistan	38928346	652860.0	60	1	i`d responded, going
1	549e992a42	Sooo SAD I will miss you here in San Diego!!!	Sooo SAD	negative	noon	21-30	Albania	2877797	27400.0	105	0	sooo sad miss san diego!!!
2	088c60f138	my boss is bullying me...	bullying me	negative	night	31-45	Algeria	43851044	2381740.0	18	0	boss bullying me...
3	9642c003ef	what interview! leave me alone	leave me alone	negative	morning	46-60	Andorra	77265	470.0	164	0	interview! leave alone
4	358bd9e861	Sons of ****, why couldn`t they put them on t...	Sons of ****,	negative	noon	60-70	Angola	32866272	1246700.0	26	0	sons ****, couldn`t put releases already bought

Figure 2: Data Preview

We observe that the preprocessing worked, and we will now work solely with the preprocessed text, disregarding the other features.

```python
# 1. Transform sentiment into 3 classes
# Example mapping: positive -> 2, neutral -> 1, negative -> 0
sentiment_mapping = {"positive": 2, "neutral": 1, "negative": 0}
train_df["sentiment_class"] = train_df["sentiment"].map(sentiment_mapping)
test_df["sentiment_class"] = test_df["sentiment"].map(sentiment_mapping)

# 2. Extract all the values from the 'processed_text' column into a list
trainval_x = train_df["processed_text"].tolist()
trainval_y = train_df["sentiment_class"].tolist()

train_x, val_x, train_y, val_y = train_test_split(trainval_x, trainval_y,
    test_size=0.25, random_state=42)

test_x = test_df["processed_text"].tolist()
test_y = test_df["sentiment_class"].tolist()
```

We then extract the embeddings using BERT, which we will use throughout this project.

Visualization using PCA

We know that the dimensionality of the data is **768**. However, we would like to visualize the data in 1D, 2D, or 3D.
To achieve this, we will use the PCA method (known in French as ACP), which allows us to project our data onto the principal components of the covariance matrix for the most meaningful visualization.

Variance Analysis

First, we want to examine how the variance evolves as a function of the number of components chosen. In particular, we wish to determine the number of components needed to capture 95% of the variance.

```python
# PCA with all components
pca = PCA()
train_embeddings_pca = pca.fit_transform(train_embeddings.numpy())
# Variance ratio per component
explained_variance_ratio = pca.explained_variance_ratio_

cumul_var = 0
i = 0
# Retrieve the components whose cumulative sum constitutes 95% of the variance
while (cumul_var < 0.95):
    cumul_var += explained_variance_ratio[i]
    i += 1

n_components = i+1
print("We should have {} components to capture {}% of the variance.".format(i+1, int(cumul_var*100)))
```

```
We should have 23 components to capture 95% of the variance.
```

We then display the raw values and the variance histogram.

```python
pca = PCA(n_components)
train_embeddings_pca = pca.fit_transform(train_embeddings.numpy())

explained_variance_ratio = pca.explained_variance_ratio_
explained_variance = pca.explained_variance_
print("Explained Variance Ratio per Component:")
print(explained_variance_ratio)
print("\nExplained Variance per Component:")
print(explained_variance)
print(f"Cumulative explained variance for 3 components:
 ↪  {sum(explained_variance_ratio):.2f}")

# Display the variance histogram
plt.figure(figsize=(8, 6))
components = np.arange(1, n_components + 1)
plt.bar(components, explained_variance_ratio, color='skyblue',
 ↪  edgecolor='black')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.title('Explained Variance Ratio per Principal Component')
plt.xticks(components)
plt.show()
```

```
    Explained Variance Ratio per Component:
[0.69273156 0.10438945 0.0352178  0.02309251 0.02007595 0.01203258
 0.01042923 0.00811969 0.00682142 0.00531265 0.00430818 0.00391379
 0.00334314 0.00318951 0.00295682 0.00272996 0.00255025 0.00215913
 0.00208065 0.00204379 0.00179613 0.00175866 0.00152435]

Explained Variance per Component:
[20.593342    3.1032622   1.0469453   0.686488    0.5968127   0.35770142
  0.31003737  0.24138017  0.20278552  0.15793316  0.12807232  0.11634817
  0.09938393  0.09481703  0.08789954  0.08115551  0.07581323  0.06418614
  0.06185301  0.06075718  0.0533948   0.05228084  0.04531556]
Cumulative explained variance for 3 components: 0.95
```

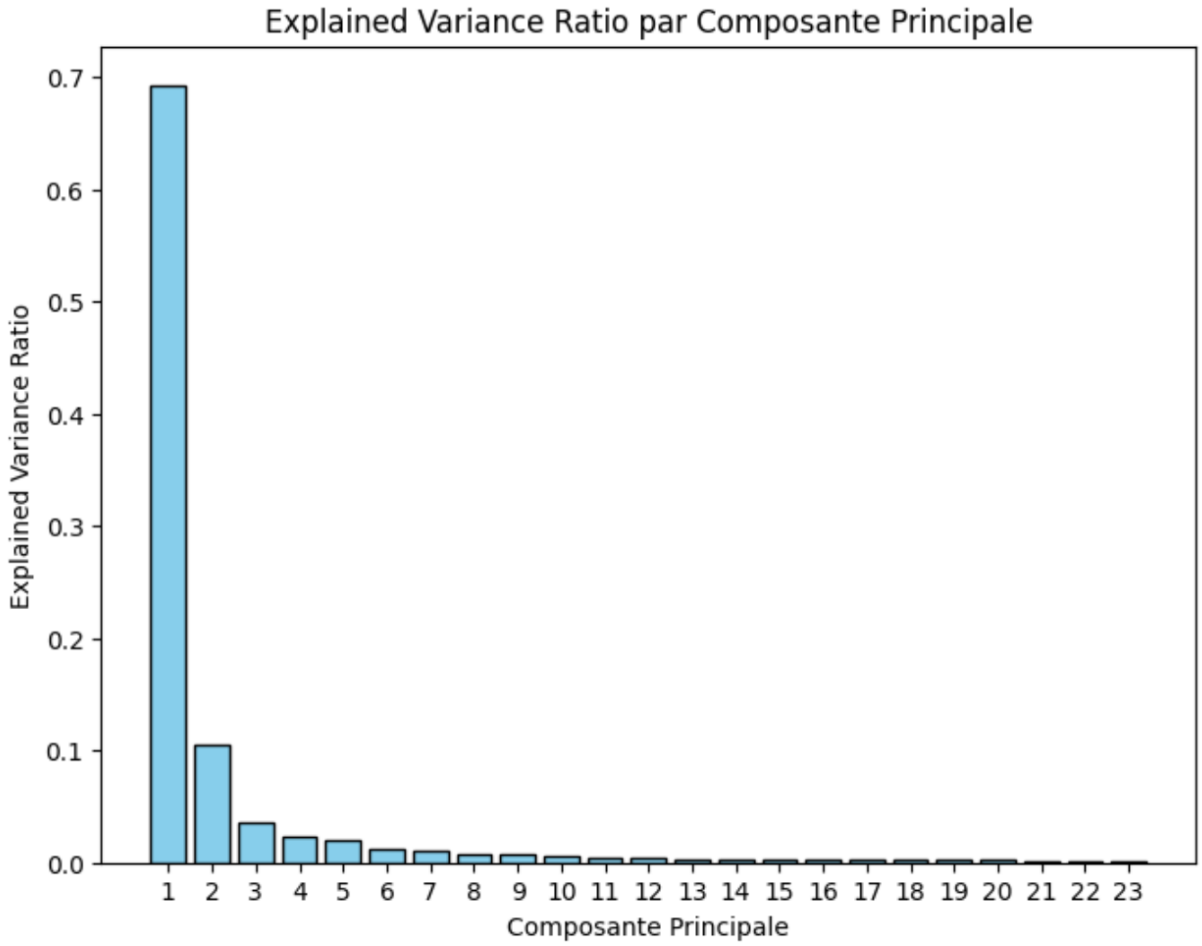

Figure 3: Histogram of Explained Variance Ratio per Principal Component

Standard PCA

We now perform PCA for each dimension to visualize the classification.

In 1D:

```python
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

n_components = 1
pca = PCA(n_components)
train_embeddings_1d = pca.fit_transform(train_embeddings.numpy())

# Histogram visualization
plt.figure(figsize=(10, 8))
plt.hist(train_embeddings_1d[:, 0], bins=30, color='skyblue',
    edgecolor='black')
plt.title("1D Histogram Visualization of PCA on the Embeddings (Training
    Set)")
```

```
plt.xlabel("Principal Component 1")
plt.ylabel("Frequency")
plt.show()
```

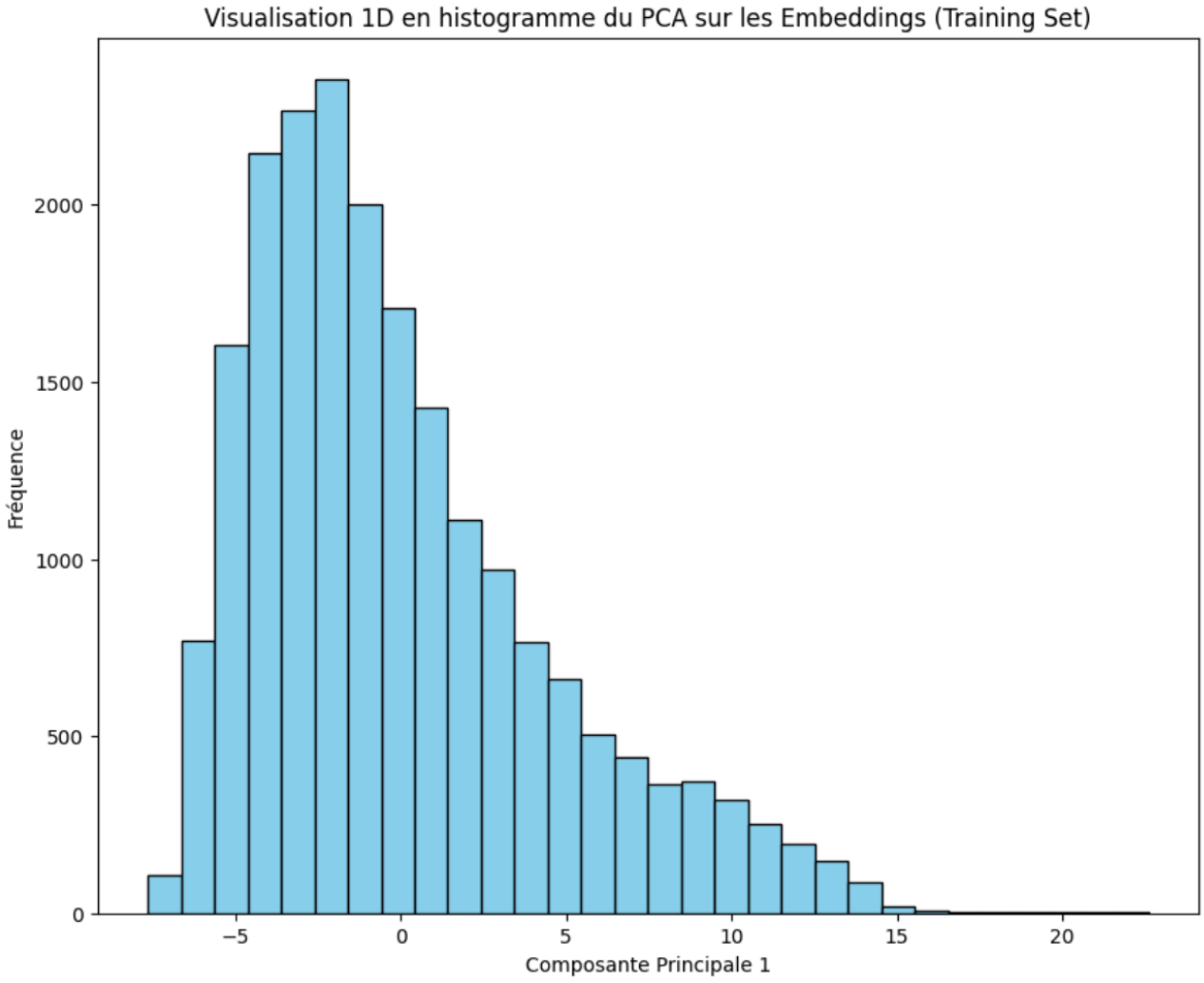

Figure 4: 1D PCA Histogram

We note that the majority of the points are located before zero, and

In 2D:

```
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA

n_components = 2
pca = PCA(n_components)
# pca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
# pca = KernelPCA(n_components=2, kernel='poly', degree=2)
# pca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
```

```
# pca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)

train_embeddings_2d = pca.fit_transform(train_embeddings.numpy())

plt.figure(figsize=(10, 8))
scatter = plt.scatter(train_embeddings_2d[:, 0], train_embeddings_2d[:, 1],
                      c=train_y, cmap='viridis', alpha=0.5)
plt.colorbar(scatter, label='Sentiment Class')
plt.title("2D Visualization of PCA on the Embeddings")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

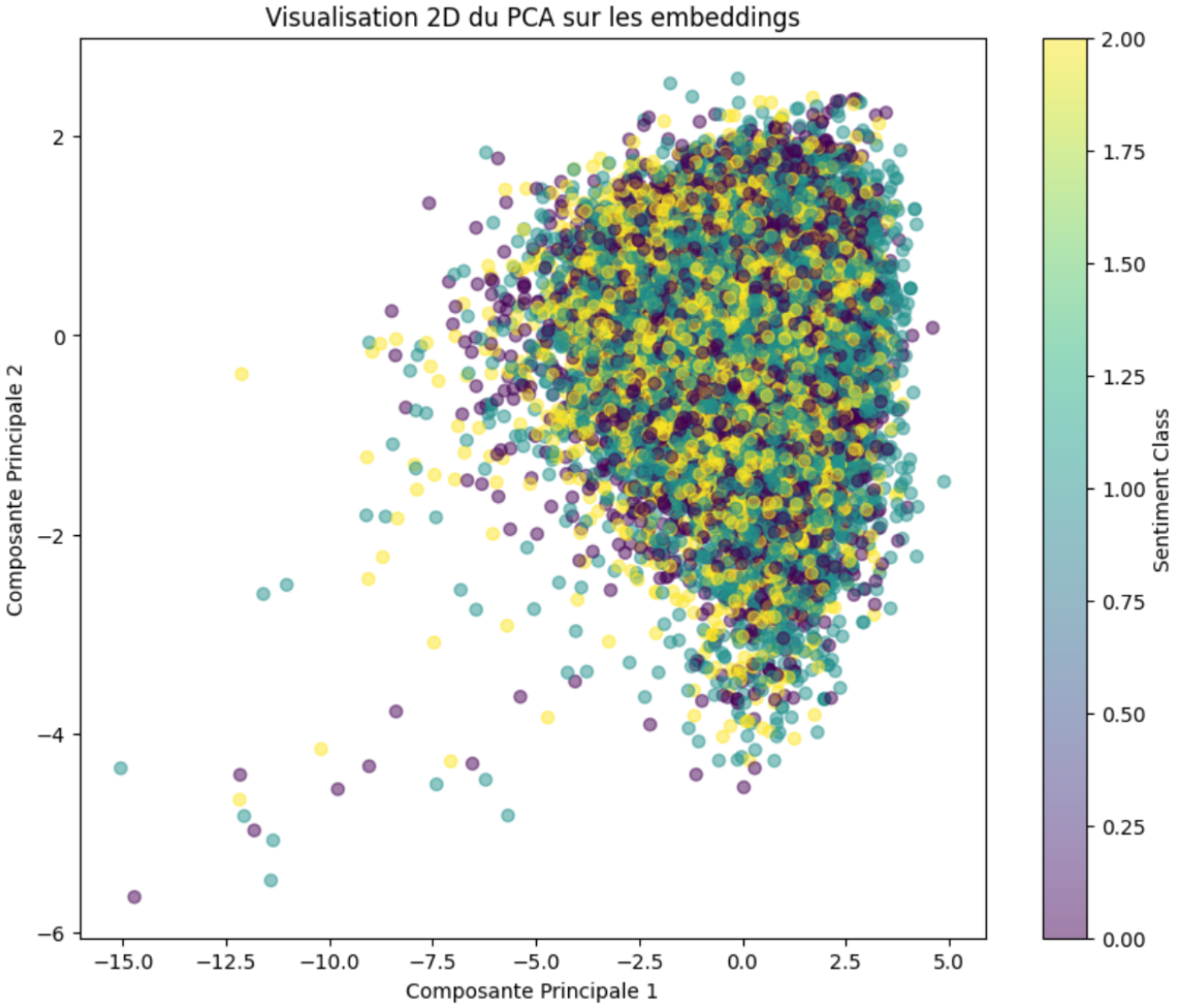

Figure 5: 2D Visualization of PCA

We observe a strong overlap of all our data, which is normal since we have projected ap-

proximately 800 dimensions onto only 2; it would have been surprising to observe a linear relationship. Therefore, it might be worthwhile to visualize the data in 3D.

In 3D:

```
from sklearn.decomposition import KernelPCA


pca = PCA(3)
train_embeddings_3d = pca.fit_transform(train_embeddings.numpy())


fig = plt.figure(1, figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d", elev=-150, azim=110)

scatter = ax.scatter(
    train_embeddings_3d[:, 0],
    train_embeddings_3d[:, 1],
    train_embeddings_3d[:, 2],
    c=train_y,
    cmap='Spectral',         # Colormap to better distinguish classes
    s=40,                    # Size of the points
    alpha=0.8                # Slight transparency
)

# Axis configuration
ax.set(
    title="Projection of the First Three PCA Dimensions",
    xlabel="Principal Component 1",
    ylabel="Principal Component 2",
    zlabel="Principal Component 3",
)

# Removal of tick labels for clarity
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.zaxis.set_ticklabels([])

# Add a colorbar to represent the classes
colorbar = fig.colorbar(scatter, ax=ax, pad=0.1, aspect=10)
colorbar.set_label('Classes')

```

```
# Display the plot
plt.show()
```

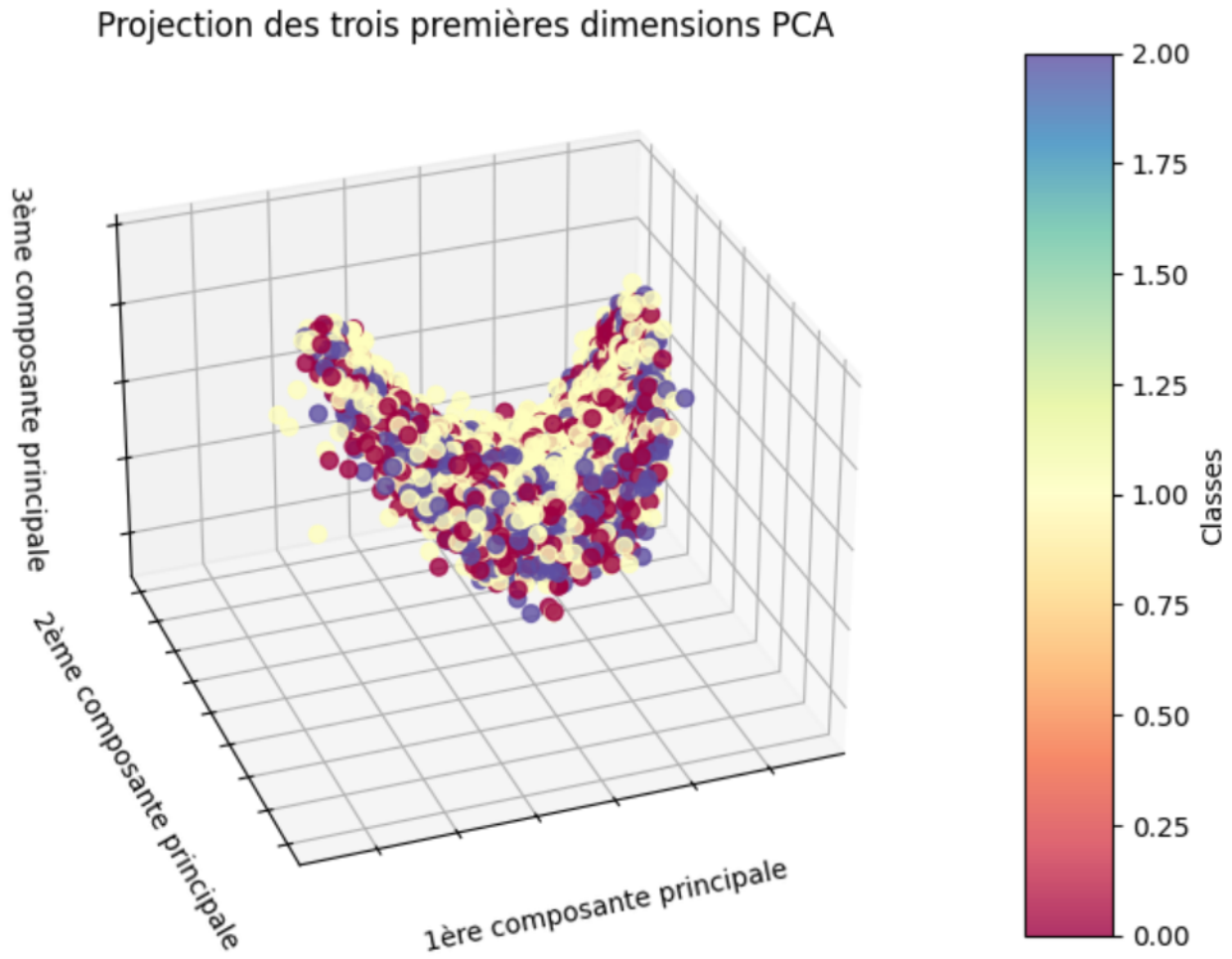

Figure 6: 3D Visualization of the PCA

It can be observed that the 3D shape is a "chip", meaning that it is concave along one axis and convex along the other. It might therefore be interesting to study the data with a PCA using a doubly polynomial kernel of order 2.

Kernel PCA

We replace the PCA definition line with:

```
pca = KernelPCA(n_components=3, kernel='poly', degree=2)
```

We then obtain:

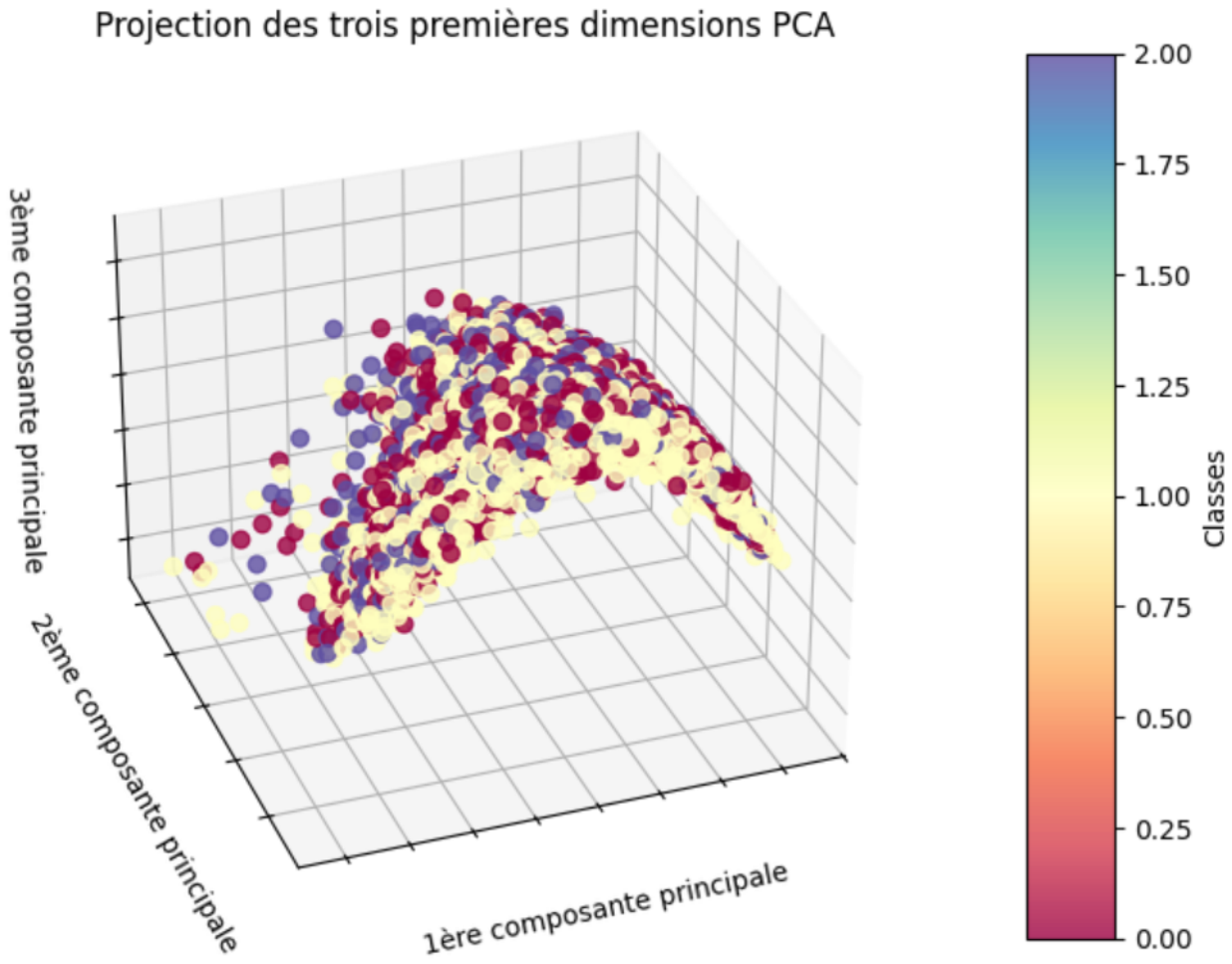

Figure 7: 3D Visualization of the KernelPCA

The points still remain very overlapped but a clearer distinction appears between the points of class 1 and the other classes.

Visualization using T-SNE

Using T-SNE, here is what we obtain.

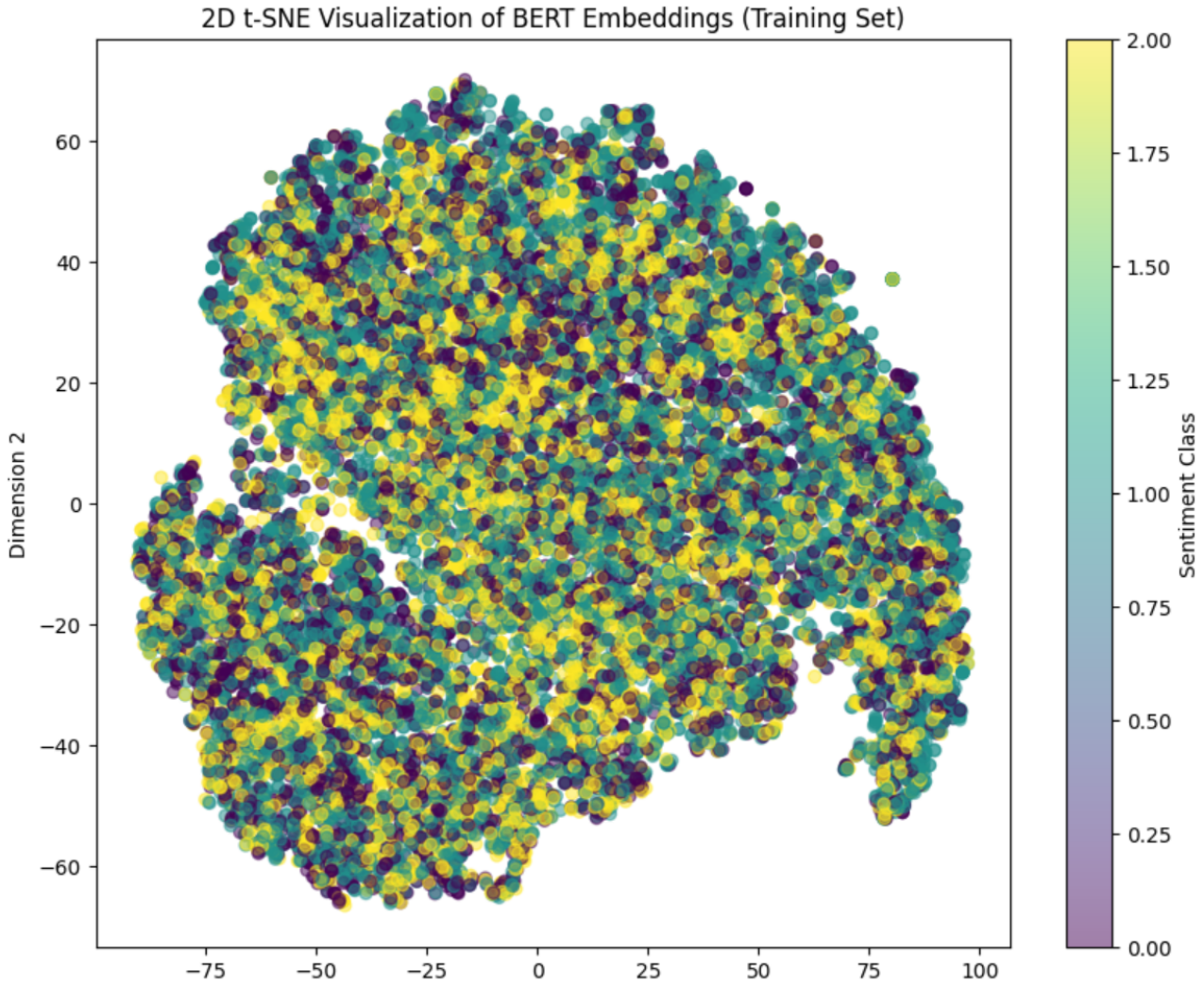

Figure 8: 2D Visualization of the TSNE

We cannot deduce anything from this figure, and the computation time is extremely long compared to PCA.

Classification using KMEANS

Starting from the PCA, we can try to run the 3-means algorithm on the processed data to attempt unsupervised classification. Using $n_{components} = 23$ to capture 95% of the variance, here are the 3-means results obtained.

```
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, DBSCAN

pca = PCA(n_components)
train_embeddings_pca = pca.fit_transform(train_embeddings.numpy())

k = 3
kmeans = KMeans(n_clusters=k, random_state=42)
clusters = kmeans.fit_predict(train_embeddings_pca)

# 3. Visualize the result
plt.figure(figsize=(8,6))
sns.scatterplot(x=train_embeddings_pca[:,0], y=train_embeddings_pca[:,1],
↪  hue=clusters, palette="viridis", s=60)
plt.title("K-means after PCA reduction")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend(title="Clusters")
plt.show()
```

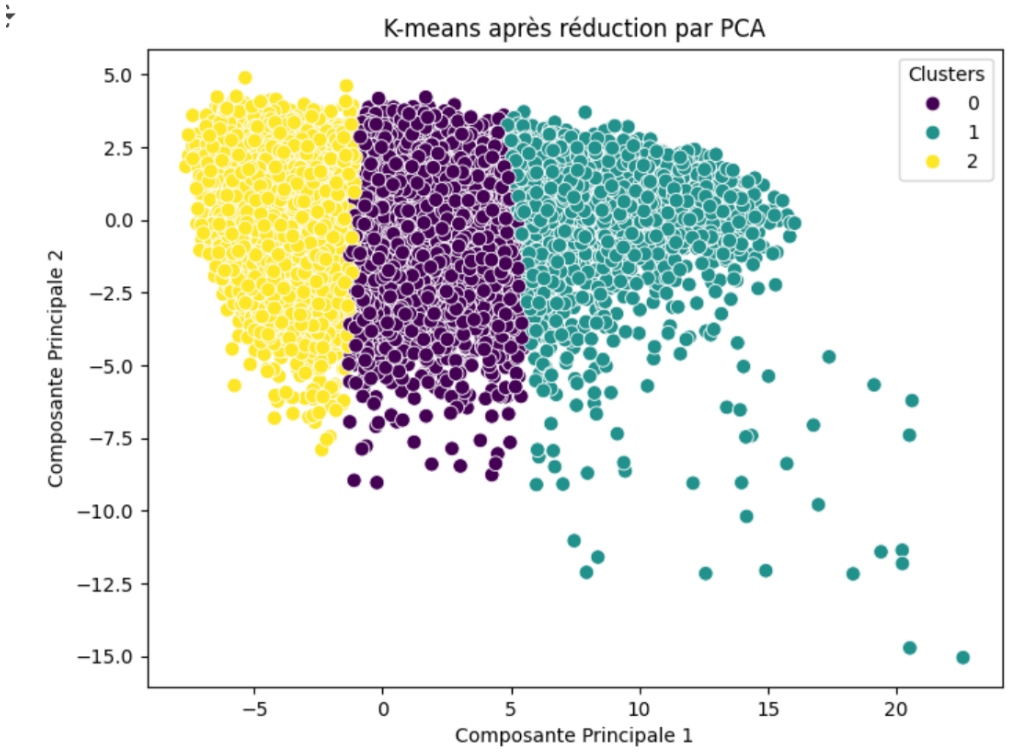

Figure 9: Visualization of the 3-means

We observe that this does not correspond at all to the classification we observed in the simple 2D PCA figure and we will verify it by calculating the accuracy.

```
def kmeans_accuracy(y_true, y_pred):
    """
    Function that calculates the accuracy of the predictions without taking
    ↪ the label into account.
    For each cluster, the majority class is chosen to determine the
    ↪ prediction.
    It is then compared with the true labels.
    """
    n = len(y_true)
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    # labels[i] represents cluster i. labels[i][j] is the number of times
    # the points in cluster i are classified as j.
    labels = [[0,0,0] for _ in range(3)]
    for i in range(n):
      labels[y_pred[i]][y_true[i]] += 1

```

```
    mapping = [np.argmax(labels[i]) for i in range(len(labels))]
    print(mapping)
    print("The mapping is: {}".format(mapping[0] != mapping[1] !=
    ↪  mapping[2]))

    score = 0
    for i in range(n):
      score += (y_true[i] == mapping[y_pred[i]])

    return score/n

kmeans_accuracy(train_y, clusters)
```

We then obtain:

```
[1, 1, 1]
The mapping is: False
0.4041727316836487
```

All points are predicted as 1, which corresponds to the neutral class that dominates the dataset and is therefore uniformly distributed in each cluster. This means that our **k-means is not capable of finding patterns** and only gives us the ratio of neutral elements in the dataset (0.40) ... By trying with a KernelPCA, we obtain exactly the same results. Therefore, PCA and clustering methods are not useful here and more complex methods will be needed to predict the points in the space.

Q1: Use of Classical Machine Learning Techniques

Data Preprocessing

The sentence embeddings extracted via BERT are used as input features for sentiment classification. Let X be the embeddings matrix and Y the associated sentiment labels. After extraction, the data is split into a training set and a test set:

```
X_train = train_embeddings.cpu().numpy()
X_test = test_embeddings.cpu().numpy()

y_train = np.array(train_y)
y_test = np.array(test_y)
```

Since machine learning models are sensitive to the scale of the data, the embeddings are normalized using a StandardScaler to improve algorithm convergence:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Training and Evaluation of the Models

A class `ModelTrainer` is defined to manage the training and evaluation of the classification models. It includes:

- Training of the model with timing.
- Evaluation of performance via accuracy and a detailed report.
- Display of a confusion matrix to visualize common errors.

```
class ModelTrainer:
    def __init__(self, model, model_name, **hyperparams):
        self.model_name = model_name
        self.model = model(**hyperparams)
        self.training_time = None
        self.results = {}

    def train(self, X_train, y_train):
        start_time = time.time()
        self.model.fit(X_train, y_train)
```

```
        self.training_time = time.time() - start_time

    def evaluate(self, X_test, y_test):
        y_pred = self.model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        report = classification_report(y_test, y_pred)
        cm = confusion_matrix(y_test, y_pred)

        self.results = {
            'accuracy': acc,
            'report': report,
            'training_time': self.training_time,
            'confusion_matrix': cm,
            'predictions': y_pred
        }

        print(f"--- {self.model_name} ---")
        print(f"Accuracy: {acc:.4f}")
        print(report)
        print(f"Training Time: {self.training_time:.2f} seconds")

        # Display the confusion matrix
        plt.figure(figsize=(6, 4))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
        ↪  xticklabels=set(y_test), yticklabels=set(y_test))
        plt.xlabel('Predicted Label')
        plt.ylabel('True Label')
        plt.title(f'Confusion Matrix - {self.model_name}')
        plt.show()

        return self.results
```

Experimentation with Different Models

Different classification models are tested to compare their performance on the BERT embeddings.

Logistic Regression

Different values for the parameter `max_iter` are tested to analyze their impact on convergence:

```
parameter = [{'max_iter':50}, {'max_iter':100}, {'max_iter':200}]
for params in parameter:
    trainer = ModelTrainer(LogisticRegression, f"LogisticRegression with
    ↪ max_iter={params['max_iter']}", **params)
    trainer.train(X_train, y_train)
    trainer.evaluate(X_test, y_test)
```

Support Vector Machine (SVM)

Several kernels (`kernel`) are explored to observe their influence on classification:

```
parameter = [
    {"kernel": "linear"},
    {"kernel": "poly", "degree": 3, "gamma": "scale", "coef0": 1},
    {"kernel": "rbf", "gamma": "scale"},
    {"kernel": "sigmoid", "gamma": "scale", "coef0": 0}
]

for params in parameter:
    trainer = ModelTrainer(SVC, f"SVM with kernel={params['kernel']}",
    ↪ **params)
    trainer.train(X_train, y_train)
    trainer.evaluate(X_test, y_test)
```

Random Forest

We tested several ensemble sizes (`n_estimators`):

```
parameter = [{'n_estimators':50}, {'n_estimators':100}, {'n_estimators':150},
↪ {'n_estimators':200}]
for params in parameter:
    trainer = ModelTrainer(RandomForestClassifier, f"RandomForestClassifier
    ↪ with n_estimators={params['n_estimators']}", **params)
    trainer.train(X_train, y_train)
    trainer.evaluate(X_test, y_test)
```

Q2: Use of an MLP for Prediction

Definition and Code

First, before building our MLP, we must extract our embeddings using BERT to be able to process our text as a 768-dimensional vector.

```
# Hyperparameters
PRETRAINED_MODEL = "bert-base-uncased"
MAX_LENGTH = 128
BATCH_SIZE = 64
NUM_CLASSES = 3
LEARNING_RATE = 2e-5
EPOCHS = 10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize tokenizer, dataset, and dataloader
tokenizer = BertTokenizer.from_pretrained(PRETRAINED_MODEL)
traindataset = TextDataset(train_x, train_y, tokenizer, MAX_LENGTH)
trainloader = DataLoader(traindataset, batch_size=BATCH_SIZE, shuffle=False)
valdataset = TextDataset(val_x, val_y, tokenizer, MAX_LENGTH)
valloader = DataLoader(valdataset, batch_size=BATCH_SIZE, shuffle=False)

bert = BertModel.from_pretrained(PRETRAINED_MODEL).to(device)
# Extract embeddings
train_embeddings = extract_embeddings(bert, trainloader, device)
train_embeddings = train_embeddings.cpu()

val_embeddings = extract_embeddings(bert, valloader, device)
val_embeddings = val_embeddings.cpu()

# Test embeddings
testdataset = TextDataset(test_x,test_y, tokenizer, MAX_LENGTH)
testloader = DataLoader(testdataset, batch_size=BATCH_SIZE, shuffle=False)

bert = BertModel.from_pretrained(PRETRAINED_MODEL).to(device)
# Extract embeddings

test_embeddings = extract_embeddings(bert, testloader, device)
test_embeddings = test_embeddings.cpu()
```

All embeddings are respectively stored in `train_embeddings` or `test_embeddings`. We now define our model.

```
criterion = nn.CrossEntropyLoss()
# Model Definition
class SentimentClassifier2(nn.Module):
    def __init__(self, hidden_sizes, output_size):
        super(SentimentClassifier2, self).__init__()

        n = len(hidden_sizes)
        self.fc_layers = nn.ModuleList()
        self.bn_layers = nn.ModuleList()
        # Fully connected layer for classification
        for i in range(n-1):
          self.fc_layers.append(nn.Linear(hidden_sizes[i],
          ↪  hidden_sizes[i+1]))
          self.bn_layers.append(nn.BatchNorm1d(hidden_sizes[i + 1]))

        self.fc_layers.append(nn.Linear(hidden_sizes[-1], output_size))
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # Pass through the fully connected layers
        n = len(self.fc_layers)
        for fc, bn in zip(self.fc_layers, self.bn_layers):
            x = fc(x)
            x = bn(x)
            x = F.gelu(x)
            x = self.dropout(x)

        x = self.fc_layers[-1](x)
        return x

# Training script

def train(model, train_loader, optimizer, epoch, log_interval=50):
    model.train()
    loss_cpu = 0
    correct = 0
    total = 0
```

```
    for batch_idx, data in enumerate(train_loader, 0):
        # Get the inputs; data is a list of [inputs, labels]
        inputs, target = data['input_ids'], data['label']
        inputs, target = inputs.cuda(), target.cuda()
        # inputs = inputs.detach()
        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, target)

        loss.backward()
        optimizer.step()
        _, predicted = torch.max(outputs.data, 1)
        loss_cpu += loss.item()
        total += target.size(0)
        correct += predicted.eq(target.data).cpu().sum()

        if batch_idx % log_interval == 0:
            print('| Epoch [%3d/%3d] Iter[%3d/%3d]\t\tLoss: %.4f Acc@1:
            ↪ %.3f%%'
                %(epoch, EPOCHS, batch_idx+1,
                    (len(train_loader)//BATCH_SIZE)+1, loss.item(),
                    ↪ 100.*correct/total))
            # n_iter = epoch * len(train_loader) + batch_idx

    return loss_cpu/len(train_loader)

def test(model, test_loader, epoch, plot=False):
    model.eval()
    correct = 0
    total = 0
    test_loss_MSE = 0

    # Initialize F1 metric
    f1_metric = MulticlassF1Score(num_classes=3, average='macro').to(device)

    with torch.no_grad():
        all_preds = []
        all_labels = []

```

```
        for batch_idx, data in enumerate(test_loader, 0):
            inputs, target = data['input_ids'], data['label']
            inputs, target = inputs.cuda(), target.cuda()

            outputs = model(inputs)
            loss = criterion(outputs, target)
            test_loss_MSE += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += predicted.eq(target.data).cpu().sum()

            # Collect predictions and labels
            all_preds.append(predicted.cpu())
            all_labels.append(target.cpu())

    # Concatenate predictions and labels
    all_preds = torch.cat(all_preds).numpy()
    all_labels = torch.cat(all_labels).numpy()

    # Calculate F1-score
    f1_score = f1_metric(torch.tensor(all_preds).to(device),
    ↪  torch.tensor(all_labels).to(device))

    # Plot confusion matrix if required
    if plot:
        cm = confusion_matrix(all_labels, all_preds)
        plt.figure(figsize=(6, 4))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
        ↪  xticklabels=["negative", "neutral", "positive"],
        ↪  yticklabels=["negative", "neutral", "positive"])
        plt.xlabel('Predicted Label')
        plt.ylabel('True Label')
        plt.title('Confusion Matrix - MLP')
        plt.show()

    # Final metrics
    test_loss_MSE = test_loss_MSE / len(test_loader)
    accuracy = 100. * correct / total
```

```
    print("\n| Validation Epoch #%d\t\t\tLoss: %.4f Acc@1: %.2f%% F1: %.2f" %
    ↪  (epoch, test_loss_MSE, accuracy, f1_score.item()))

    return test_loss_MSE, accuracy, f1_score.item()
```

Compared to the initial code provided, here are the different choices we made.

1. We chose to have multiple layers whose topology is given by the array `hidden_size`, the number of neurons in layer i is given by `hidden_sizes[i]`.
2. After each hidden layer, the activation function is **gelu**. It is defined as $GELU(x) = x\phi(x)$ with ϕ being a normal distribution. This function preserves the properties of ReLU while retaining relationships in the embedding that are lost by ReLU.
3. After each layer, we apply a **dropout of 0.2**, which disables a portion of the neurons at each step and helps prevent loss plateaus and overfitting.
4. We use cross-entropy because it is the classical loss function for classification problems.
5. We use the F1-score. This metric quantifies the number of false positives and false negatives through precision and recall. It is important when trying to prevent negative texts from being classified as positive for moderation. A good value is around **0.65**. In addition, we also display the confusion matrix.
6. We normalize the data at each layer's output.

The training code remains the same as initially provided with the few changes mentioned.

```
print ("Let us Train.")

EPOCHS = 40
input_size = len(train_embeddings[0])
output_size = 3
hidden_sizes = [input_size, 512, 128, 16]

model = SentimentClassifier2(hidden_sizes, output_size).to(device)
model_test = SentimentClassifier2(hidden_sizes, output_size).to(device)

best_error = float('inf')
LEARNING_RATE = 1e-4

from torch.optim.lr_scheduler import ReduceLROnPlateau, StepLR

traindataset = EmbeddingDataset(train_embeddings, train_y)
```

```
trainloader = DataLoader(traindataset, batch_size=BATCH_SIZE, shuffle=True)
valdataset = EmbeddingDataset(val_embeddings, val_y)
valloader = DataLoader(valdataset, batch_size=BATCH_SIZE, shuffle=False)

optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE,
↪  weight_decay=1e-4)
# lr_scheduler =  StepLR(optimizer, step_size=20, gamma=0.1)
lr_scheduler = ReduceLROnPlateau(optimizer, 'min', patience=7)

best_error = float('inf')
train_history = []
val_history = []
print('-------------------------------------------------------------------------------
print('3/ training model 1')
print('-------------------------------------------------------------------------------

iter = 0
for epoch in range(EPOCHS):
    loss = train(model, trainloader, optimizer, epoch)
    train_history.append(loss)

    val_loss, acc, _ = test(model, valloader, epoch)
    val_history.append(val_loss)

    lr_scheduler.step(val_loss)
    print('lr =', get_lr(optimizer))

    if val_loss < best_error:
        best_error = val_loss
        print('Best RMSE is of : ' + str(best_error), 'for epoch :', epoch,
        ↪  'ERROR TEST = ', val_loss)
        # model_test.parameters() = model.state_dict()
        model_test.load_state_dict(model.state_dict(), strict=True)
        torch.save(model_test.state_dict(), "best_model.pth")  # Save the
        ↪  model

print ("Training Done!")
```

“‘latex

Hyperparameter Choices and Performance

With all the choices mentioned above, we observed several phenomena to be mitigated.

1. Adding new layers marginally improves the model's effectiveness. To have an effective contribution, we chose that **each layer would be a power of 2** and **the batch size would be 32**. hidden_sizes = [input_size, 512, 128, 16]

2. The loss function reaches a plateau after N=25 epochs. To mitigate this, we added a **dropout of 0.2**, we **used a ReduceLROnPlateau scheduler**, we **normalized the data at each layer's output** and we chose a learning rate of $\eta = 10^{-4}$ to favor convergence toward an optimum. Additionally, we chose EPOCHS = 50 to avoid overfitting.

3. The validation loss function differs from the training loss, which is characteristic of overfitting. The dropout once again helps improve performance in this case.

Results

We then obtain for the previous choices:

```
    Let us Train.
-----------------------------------------------------------------------------------
3/ training model 1
-----------------------------------------------------------------------------------
| Epoch [  0/ 40] Iter[  1/  6]                Loss: 1.0953 Acc@1: 34.375%
| Epoch [  0/ 40] Iter[ 51/  6]                Loss: 1.1323 Acc@1: 35.049%
| Epoch [  0/ 40] Iter[101/  6]                Loss: 1.0297 Acc@1: 39.851%
| Epoch [  0/ 40] Iter[151/  6]                Loss: 0.9967 Acc@1: 42.074%
| Epoch [  0/ 40] Iter[201/  6]                Loss: 1.0124 Acc@1: 44.364%
| Epoch [  0/ 40] Iter[251/  6]                Loss: 0.9834 Acc@1: 45.667%
| Epoch [  0/ 40] Iter[301/  6]                Loss: 0.9906 Acc@1: 46.865%

| Validation Epoch #0                       Loss: 1.0187 Acc@1: 46.10% F1:
 ↪  0.45
lr = 0.0001
Best RMSE is of : 1.0187300780305155 for epoch : 0 ERROR TEST =
 ↪  1.0187300780305155
| Epoch [  1/ 40] Iter[  1/  6]                Loss: 0.9204 Acc@1: 60.938%
| Epoch [  1/ 40] Iter[ 51/  6]                Loss: 1.0690 Acc@1: 55.300%
| Epoch [  1/ 40] Iter[101/  6]                Loss: 0.9006 Acc@1: 55.554%
| Epoch [  1/ 40] Iter[151/  6]                Loss: 0.9869 Acc@1: 55.867%
| Epoch [  1/ 40] Iter[201/  6]                Loss: 1.0376 Acc@1: 56.172%
| Epoch [  1/ 40] Iter[251/  6]                Loss: 0.9264 Acc@1: 56.431%
```

```
| Epoch [  1/ 40] Iter[301/  6]                Loss: 0.8589 Acc@1: 56.785%

...

| Validation Epoch #39                        Loss: 0.7976 Acc@1: 64.32% F1:
↪  0.64
lr = 1.0000000000000002e-06
Training Done!
```

The evolution of the Loss curve is given below.

```
plt.figure(figsize=(8, 6))
plt.plot(train_history[4:], label='Train Loss', marker='o')
plt.plot(val_history[4:], label='Validation Loss', marker='o')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training and Validation Loss over Epochs")
plt.legend()
plt.grid(True)
plt.show()
```

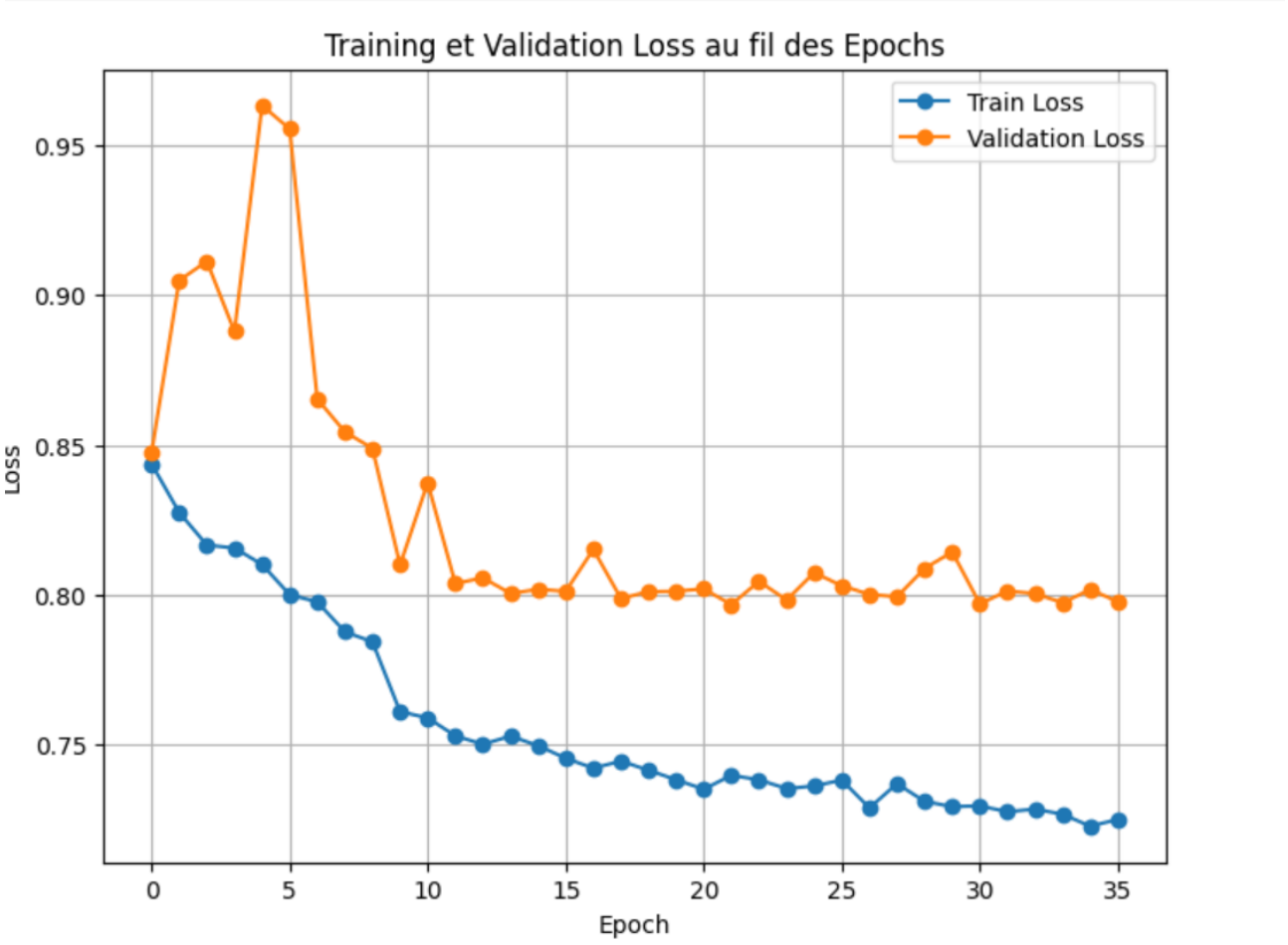

Figure 10: Evolution of the loss curve over Epochs (40 epochs)

We then test the model on the validation data.

```
testdataset = TextDataset(test_x,test_y, tokenizer, MAX_LENGTH)
testloader = DataLoader(testdataset, batch_size=BATCH_SIZE, shuffle=False)

bert = BertModel.from_pretrained(PRETRAINED_MODEL).to(device)
# Extract embeddings

test_embeddings = extract_embeddings(bert, testloader, device)
test_embeddings = test_embeddings.cpu()

print('==================== ')
print('Test set = ')
testdataset = EmbeddingDataset(test_embeddings, test_y)
testdataset = DataLoader(testdataset, batch_size=BATCH_SIZE, shuffle=True)
test_loss, accu, f1 = test(model, testdataset, epoch, True)
print('==================== ')
```

```
print("Test_loss: {}, Accu: {}, F1-score: {}, Taux d'erreur critique:
↪  ??".format(test_loss, accu, f1))
```

We then obtain these indicators:

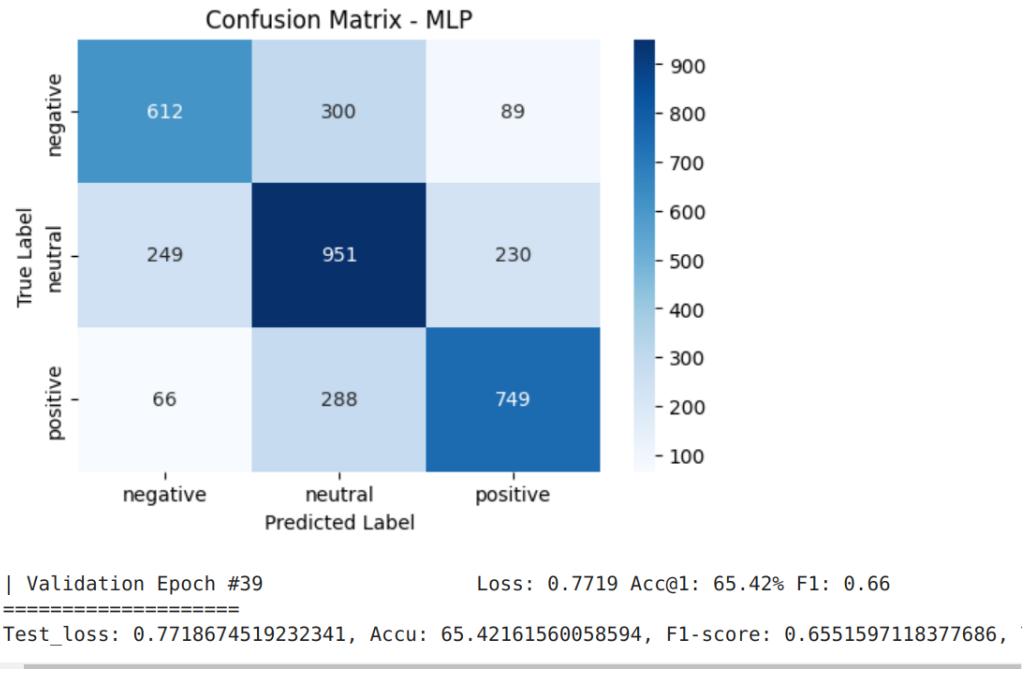

Figure 11: MLP Results on the validation data

We have a fairly good F1-score of **F1-score = 0.65**. However, despite all our trials, the maximum accuracy is **Acc = 65.42%**. This is rather good since it approaches the accuracy of the best models, as we will see in the following sections. Finally, we note that 89 negative texts are classified as positive, which is low compared to the amount of data but too high to use this predictor as a moderation tool.

Note: We obtained an accuracy of 65.7% after several training trials by changing the random seed, and we consider the MLP to have this accuracy for the remainder.

Q3: Performance Analysis and Selection of the Best Model

We analyze the performance of the sentiment classification models using three main metrics:

- **Accuracy**: percentage of correct classifications.
- **Average F1-score**: balance between precision and recall.
- **Critical Error Rate**: proportion of completely inverted predictions (e.g., positive classified as negative and vice versa). This score, which may initially seem exotic, is considered important because it is thought to be much less serious to classify a text as neutral knowing that it is positive or negative (or vice versa) than to classify a text as positive when it is negative (or vice versa).

Model Performance

The results are presented in the table below:

Model	Accuracy	F1-score	Critical Error Rate	Time (s)
Logistic Regression (50)	63.6%	0.64	4%	2.01
Logistic Regression (100)	65.3%	0.65	4%	4.64
Logistic Regression (200)	65.6%	0.66	4%	5.48
SVM (linear)	64.2%	0.64	5%	2806.39
SVM (poly)	64.8%	0.65	4%	236.24
SVM (RBF)	63.5%	0.63	4%	212.49
SVM (sigmoid)	34.0%	0.34	25%	153.29
Random Forest (50 trees)	53.7%	0.53	6%	21.06
Random Forest (100 trees)	54.2%	0.53	5%	41.76
Random Forest (150 trees)	55.6%	0.54	5%	62.47
Random Forest (200 trees)	55.0%	0.54	5%	84.14
MLP	65.7%	0.66	4.75%	N/A

Table 1: Comparison of model performance

Comparative Analysis

Accuracy The MLP model achieves the highest accuracy with **65.7%**, slightly surpassing logistic regression (**65.6%**). The SVM and random forest models show lower performance, except for the `poly` kernel which reaches **64.8%**.

F1-score The evolution of the F1-score follows that of accuracy, with the MLP and logistic regression at the top (**0.66**), and SVM `poly` just behind (**0.65**). The random forests remain behind (**0.54**).

Critical Error Rate The SVM `sigmoid` remains the worst choice with a critical error rate of **25%**. The MLP displays a slightly higher rate than logistic regression (**4.75%** vs. **4%**), which is still within the same range of performance.

Training Time The training time of the MLP is not provided here, but it strongly depends on the network size and available computational power. In contrast, the linear and poly SVMs are the most time-consuming (**2806s** and **236.24s** respectively), while logistic regression remains very fast.

Selection of the Optimal Model

Based on these results, two models stand out:

- **MLP**, which offers the highest accuracy (**65.7%**) and an F1-score identical to logistic regression (**0.66**).
- **Logistic Regression with max_iter=200**, which has a very close accuracy (**65.6%**), a lower critical error rate (**4%**) and a very short training time.

Thus, the MLP is a promising alternative, although it may require more computational resources depending on its configuration.

Conclusion

Logistic regression and the MLP are the two best candidates for this task. If computational constraints are limited, the MLP can be an interesting choice to obtain a slight improvement in performance. However, logistic regression remains an excellent compromise between accuracy, speed, and robustness.

Q4: Comparison with the Performance of an LLM

We aim to compare the performance of our best model with that of an LLM. We therefore import the **Roberta** model from Hugging Face, which is specialized in sentiment classification, in order to compare its performance on the test data with our model.

```python
from transformers import pipeline
import numpy as np
from sklearn.metrics import accuracy_score, classification_report

label_mapping = {"negative": 0, "neutral": 1, "positive": 2}

# Initialization of an LLM specialized in sentiment detection via a
# Hugging Face pipeline. Here we load Roberta
llm_classifier = pipeline(
    "sentiment-analysis",
    model="cardiffnlp/twitter-roberta-base-sentiment-latest",
    tokenizer="cardiffnlp/twitter-roberta-base-sentiment-latest"
)

# Evaluation of the LLM on the preprocessed texts
llm_preds = []
for text in test_x:
    result = llm_classifier(text)
    predicted_label = result[0]["label"].lower()
    llm_preds.append(label_mapping[predicted_label])

llm_preds = np.array(llm_preds)
y_test_np = np.array(test_y)

# Calculate the model's accuracy
llm_accuracy = accuracy_score(y_test_np, llm_preds)
llm_report = classification_report(
    y_test_np, llm_preds, target_names=["negative", "neutral", "positive"]
)

print("=== LLM Sentiment Classifier Performance ===")
print("LLM Accuracy: {:.4f}".format(llm_accuracy))
print("\nLLM Classification Report:")
print(llm_report)
```

We then obtain:

```
=== LLM Sentiment Classifier Performance ===
LLM Accuracy: 0.7043

LLM Classification Report:
              precision    recall  f1-score   support

    negative       0.71      0.72      0.71      1001
     neutral       0.71      0.59      0.65      1430
    positive       0.70      0.84      0.76      1103

    accuracy                           0.70      3534
   macro avg       0.70      0.72      0.71      3534
weighted avg       0.70      0.70      0.70      3534
```

It is then very clear that **the MLP's accuracy is very close to that of the LLM** since if the LLM represents the upper limit, **the maximum accuracy would be 70%** whereas the MLP achieves **65.7%**. This maximum limit therefore seems to be determined by the data preprocessing, and reflects the low variance distribution during the principal component decomposition.

Model	**Accuracy**	**F1-score**	**Critical Error Rate**	**Time (s)**
Roberta-LLM	70%	0.71	N/A %	N/A

Table 2: Performance of the LLM on the test set

Q5: Use of BERT

Using BERT embeddings allows the transformation of text into representations that are practical to use. The objective is to obtain vectors that incorporate the meaning of words based on their context, thus facilitating machine learning on natural language processing tasks.

BERT generates contextualized embeddings, which means that the representation of a word depends on the context in which it appears. Unlike classical approaches such as Word2Vec or TF-IDF, this allows for better handling of polysemy and the subtleties of language.

The model is pre-trained on large amounts of data and thus integrates advanced linguistic and semantic knowledge, which is very useful for handling complex cases of language form such as double negation.

Finally, the use of embeddings also reduces dimensionality and noise compared to raw textual representations. Classical methods, such as one-hot encoding, produce very long and sparse vectors, whereas BERT embeddings provide dense and compact vectors, which are more effective for machine learning.

Q6: How BERT Works

Input Representation

The input is composed of three types of embeddings:

- **Token Embeddings**: Each word or sub-word (via WordPiece) is converted into a learned vector.
- **Segment Embeddings**: When the input contains two sentences (for the NSP task), an embedding indicates which sentence each token belongs to.
- **Positional Embeddings**: Since the Transformer does not understand the order of words, these embeddings add information about the position of each token.

These three embeddings are summed before being fed into the model.

BERT Architecture

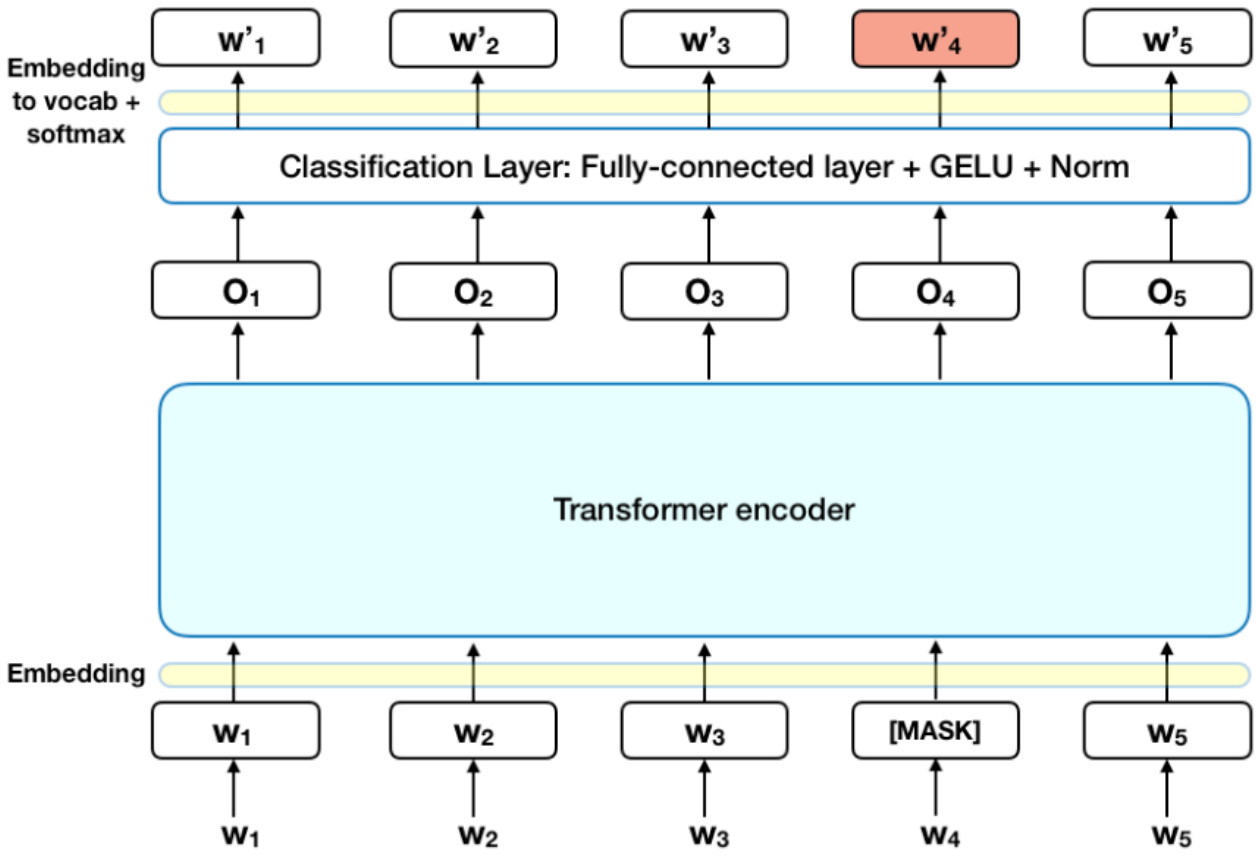

Figure 12: BERT Architecture

BERT consists of several layers of Transformer encoders (12 for BERT-Base, 24 for BERT-Large). Each layer includes:

- **Multi-Head Self-Attention**: Each word can interact with all others in the sequence to capture the global context. Multiple attention heads are used to learn different relationships.
- **Feed-Forward Network**: After self-attention, each word passes through a simple neural network (two linear layers with GELU activation).
- **Layer Normalization and Residual Connections**: Each layer is stabilized by normalization and residual connections to improve learning.

BERT Pre-training

BERT learns from two main tasks:

- **Masked Language Modeling (MLM)**: 15% of the words are masked, and the model must predict them using their context.
- **Next Sentence Prediction (NSP)**: The model receives two sentences and must guess whether they logically follow one another.

Bidirectionality

Unlike models that read text in one direction, BERT simultaneously captures both left and right contexts, allowing it to better understand the overall meaning of a text.

Q7: Fine-tuning RoBERTa with LoRA for Sentiment Classification

In this section, we train a sentiment classification model based on **RoBERTa**, a pre-trained language model, using the **LoRA** (Low-Rank Adaptation) technique. LoRA allows for efficient model adaptation without requiring an update of all of the model's weights, thereby reducing computational complexity.

Model Setup

We use the `roberta-base` model from the `transformers` library, which is specialized in natural language processing. We apply LoRA to limit the number of parameters adjusted during training.

```python
import torch
import numpy as np
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from peft import LoraConfig, get_peft_model, TaskType

# Load the model and tokenizer
MODEL_NAME = "roberta-base"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSequenceClassification.from_pretrained(MODEL_NAME,
 ↪  num_labels=3)

# Apply LoRA
lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS,
    inference_mode=False,
    r=8,
    lora_alpha=32,
    lora_dropout=0.1
)
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

Data Preparation

We use the same dataset as we used previously for the MLP

```python
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        encoded = self.tokenizer(
            self.texts[idx],
            truncation=True,
            padding="max_length",
            max_length=self.max_length,
            return_tensors="pt"
        )
        return {key: val.squeeze(0) for key, val in encoded.items()} |
        ↪ {"labels": torch.tensor(self.labels[idx], dtype=torch.long)}
```

We define the training, validation, and test sets:

```python
train_dataset = SentimentDataset(train_x, train_y, tokenizer)
val_dataset = SentimentDataset(val_x, val_y, tokenizer)
test_dataset = SentimentDataset(test_x, test_y, tokenizer)
```

Model Training

We use `Trainer` from the `transformers` library to train our model with the following arguments:

```python
from transformers import TrainingArguments, Trainer
from sklearn.metrics import accuracy_score

# Define the metrics
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
```

```
    return {"accuracy": accuracy_score(labels, preds)}

# Hyperparameter configuration
training_args = TrainingArguments(
    output_dir="./lora_roberta_sentiment",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_steps=10,
    load_best_model_at_end=True,
)

# Initialize the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer,
)

# Train the model
import time
start_time = time.time()
trainer.train()
training_time = time.time() - start_time
print(f"\nTraining completed in {training_time:.2f} seconds")
```

Evaluation and Results

We evaluate the model on the validation and test sets by analyzing the classification metrics.

```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

```python

# Evaluation on the validation set
eval_results = trainer.evaluate()
print("\n--- Evaluation Results on Validation ---")
print(eval_results)

# Predictions on the test set
preds_output = trainer.predict(test_dataset)
predictions = np.argmax(preds_output.predictions, axis=-1)

accuracy = accuracy_score(test_y, predictions)
report = classification_report(test_y, predictions, target_names=["negative",
↪  "neutral", "positive"])
cm = confusion_matrix(test_y, predictions)

print("\n--- Classification Report on Test ---")
print(report)
print(f"Accuracy: {accuracy:.4f}")

plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=["negative", "neutral", "positive"],
            yticklabels=["negative", "neutral", "positive"])
plt.xlabel("Prediction")
plt.ylabel("Actual")
plt.title("Confusion Matrix - RoBERTa with LoRA (Test)")
plt.show()
```

Model	Accuracy	F1-score	Critical Error Rate	Time (s)
Fine-Tuning	74.22%	0.74	2.96%	1134

Table 3: Performance of the Fine-Tuning on the test set

Conclusion

Thus, we observed that the best performance is achieved through fine-tuning the language model, allowing for better adaptation to the sentiment classification task. However, accuracy remains improvable and could be further optimized by exploring advanced text preprocessing strategies, such as improving data cleaning, expanding contractions or integrating specific linguistic features.