DS 677 Project

Emotion Predictor

(Reference: TorchMoji)

Project Group:

Dinesh Lankepillewar

Paresh Rahate

Ojaswi Ghate

Demo:

Part 1 : https://youtu.be/gyHhkvChRU8

Part 2 :https://youtu.be/HmGPN4vBrCM

Code:

https://github.com/testUser0302/EmotionPredictor.git

Purpose

This project aimed at unraveling the intricate tapestry of emotions woven into texts on social media or reviews. We have taken the reference of TorchMoji and it is developed as a pyTorch implementation of the DeepMoji model, the project leverages a colossal dataset of 1.2 billion tweets adorned with emojis. Its objective is to comprehend how language, specifically tweets, becomes a canvas for the expression of emotions. Through transfer learning, the model achieves state-of-the-art performance on various emotion-related text modeling tasks. In a similar way as TorchMoji, we implemented three different independent models which can predict the emotions of the text and categorize it into Positive, Negative or Unknown and to identify which model gives better prediction.
The project is focused on understanding the emotional nuances within text data, particularly on social media posts.
Consider an example of a social media post.

Text: "Just finished reading the latest book! Absolutely loved it, but couldn't put it down.
In this example, all three models predict a positive emotion for the social media post. This could be due to the use of positive words like "loved,"

This example illustrates how three independent models might categorize emotions in different social media posts. The actual predictions would depend on the training data, the specific features the models learned to recognize, and the nuances of the text.

Introduction:

In the fast-paced realm of social media, understanding the emotional undertones of textual content is crucial. TorchMoji addresses this by delving into the emotive landscape of tweets. By training on an extensive dataset, the model endeavors to decipher the nuanced ways in which users express their emotions through language. Similarly, we have implemented three models which will help to predict the emotion from the text.

Methodology:

The core methodology involves implementing the model in pyTorch. The model undergoes training on a dataset rich in texts, each with different vocab. Transfer learning is employed to enable the model to generalize its understanding of emotions, paving the way for robust performance across diverse text-based emotional analysis tasks.

Implementation Approach and Details:

The code is implemented for an emotion analysis model using various deep learning methodologies in PyTorch. It will include a custom dataset class, data preprocessing, model definition, training loop, and evaluation on a test set. The code also demonstrates predicting emotions for sample inputs using the trained model.

As a team of 3, we are planning to implement 3 independent models on the same dataset to predict the emotions.

We have used the contents we learnt till the time -
Regularization, Dropout, Initialization
CNNs
Attention Mechanisms - Transformers
Word Embeddings - May try to use this approach as well

Challenges:

Source: https://github.com/huggingface/torchMoji
Attempts to adapt TorchMoji to any of the PyTorch versions supported by Colab were unsuccessful. Despite exploring various strategies to make it compatible with these newer versions, none yielded positive results.

Implementation

We together have discussed the structure of the complete program and will be modifying the models independently.

Below are the steps used for implementing the model -

Model 1

1. Import necessary libraries such as torch, torch.nn, torch.optim, torch.utils.data, sklearn, and random.
2. Create a custom dataset class (EmotionDataset) that inherits from torch.utils.data.Dataset. Implement the init, len, and getitem methods to handle the dataset.
3. Define a function (collate_batch) to pad sequences within each batch using pad_sequence from torch.nn.utils.rnn.
4. Implement a function (read_data_from_csv) to read data from a CSV file, extracting texts and labels.
5. Create a class (EmotionAnalysisModel) for the emotion analysis model, which inherits from torch.nn.Module. Define layers: embedding layer, LSTM layer, linear layer, and a dropout layer for regularization.
6. Read data from the CSV file, split it into training and testing sets using train_test_split, and create a vocabulary with a mapping from words to indices (word_to_idx).
7. Instantiate training and testing datasets using the defined classes and update them with the word indices.
8. Create DataLoader objects for training and testing using torch.utils.data.DataLoader.
9. Initialize the emotion analysis model, specifying parameters such as vocabulary size, embedding dimension, hidden dimension, and output dimension.
10. Define a loss function (nn.CrossEntropyLoss) and an optimizer (optim.Adam) for training the model.
11. Train the model for a specified number of epochs, iterating through batches in the training data. Zero the gradients, forward pass, compute loss, backward pass, and optimize the model parameters.
12. Evaluate the trained model on the test set, computing accuracy using accuracy_score from sklearn.
13. Provide sample inputs, preprocess them by converting words to indices using the word_to_idx mapping. Pass the preprocessed inputs through the trained model, interpret the output, and map the predicted class to emotion labels.
14. Print the test accuracy and predicted emotions for each sample input.

Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from torch.nn.utils.rnn import pad_sequence
import random

# Define a simple dataset class
class EmotionDataset(Dataset):
    def __init__(self, texts, labels, word_to_idx):
        self.texts = texts
        self.labels = labels
        self.word_to_idx = word_to_idx

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

    def __getitem__(self, idx):
        text = [self.word_to_idx[word] for word in
self.texts[idx].split()]

        # Try to convert the label to an integer, handle non-numeric
labels
        try:
            label =
torch.as_tensor(int(self.labels[idx])).clone().detach()
        except ValueError:
            # Handle the case where the label is not a valid integer
            label = torch.as_tensor(0)  # Set a default value or handle it
as appropriate

        return {'text': torch.LongTensor(text), 'label': label}

def collate_batch(batch):
    texts = [item['text'] for item in batch]
    labels = [item['label'] for item in batch]
    # Pad sequences to the same length within each batch
    padded_texts = pad_sequence(texts, batch_first=True, padding_value=0)

    return {'text': padded_texts, 'label': torch.stack(labels)}

def read_data_from_csv(file_path):
    texts = []
    labels = []
    with open(file_path, 'r') as file:
        next(file)  # Skip header if exists
        for line in file:
            parts = line.strip().split(',')
            if len(parts) >= 2:
                texts.append(parts[0])
                labels.append(parts[1].strip('\"'))
    return texts, labels

# Text preprocessing
class EmotionAnalysisModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(EmotionAnalysisModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(0.5)  # Add dropout for regularization

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        lstm_out = lstm_out[:, -1, :]
        lstm_out = self.dropout(lstm_out)  # Apply dropout
        out = self.fc(lstm_out)
        return out

# Read data from CSV file
file_path = '/content/gdrive/MyDrive/DS677 - Fall 23 - DL Project -
Paresh, Ojaswi, Dinesh/TextAndEmotions.csv'
texts, labels = read_data_from_csv(file_path)

# Create vocabulary and word_to_idx mapping
```

```
vocab = set(' '.join(texts).split())
word_to_idx = {word: idx + 1 for idx, word in enumerate(vocab)}  # Add 1
to reserve index 0 for padding

# Update the dataset and pad sequences
train_texts, test_texts, train_labels, test_labels =
train_test_split(texts, labels, test_size=0.2, random_state=42)

train_dataset = EmotionDataset(train_texts, train_labels, word_to_idx)
test_dataset = EmotionDataset(test_texts, test_labels, word_to_idx)

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_batch)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False, collate_fn=collate_batch)

# Update the model to handle the new word indices
model = EmotionAnalysisModel(vocab_size=len(word_to_idx) + 1,
embedding_dim=50, hidden_dim=100, output_dim=len(set(labels))).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 50
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch in train_loader:
        text, label = batch['text'].to(device), batch['label'].to(device)
        optimizer.zero_grad()
        output = model(text)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    print(f'Epoch {epoch + 1}/{epochs}, Loss: {total_loss /
len(train_loader)}')
```

```
model.eval()
all_predictions = []
all_labels = []

with torch.no_grad():
    for batch in test_loader:
        text, label = batch['text'].to(device), batch['label'].to(device)
        output = model(text)
        predictions = torch.argmax(output, dim=1)
        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(label.cpu().numpy())

accuracy = accuracy_score(all_labels, all_predictions)
print(f'Test Accuracy: {accuracy * 100:.2f}%')
```

Test Accuracy of The Model

```
Epoch 1/50, Loss: 4.547698232862684
Epoch 2/50, Loss: 2.958804806073507
Epoch 3/50, Loss: 1.4416541390948825
Epoch 4/50, Loss: 0.6148142417271932
Epoch 5/50, Loss: 0.4744644496175978
Epoch 6/50, Loss: 0.4506297939353519
Epoch 7/50, Loss: 0.45748965938886005
Epoch 8/50, Loss: 0.40766768985324436
Epoch 9/50, Loss: 0.4111311700608995
Epoch 10/50, Loss: 0.406077245871226
Epoch 11/50, Loss: 0.40343988604015774
Epoch 12/50, Loss: 0.3970780074596405
Epoch 13/50, Loss: 0.4070357580979665
Epoch 14/50, Loss: 0.4112670554055108
Epoch 15/50, Loss: 0.40231814980506897
Epoch 16/50, Loss: 0.40141647722986007
Epoch 17/50, Loss: 0.4043105145295461
Epoch 18/50, Loss: 0.410246878862381
Epoch 19/50, Loss: 0.3874448438485463
Epoch 20/50, Loss: 0.40347252951727974
Epoch 21/50, Loss: 0.3785329858462016
Epoch 22/50, Loss: 0.38605739838547176
Epoch 23/50, Loss: 0.39331305358144975
Epoch 24/50, Loss: 0.37998779283629525
Epoch 25/50, Loss: 0.3797275788254208
Epoch 26/50, Loss: 0.3898761636681027
Epoch 27/50, Loss: 0.3883558064699173
Epoch 28/50, Loss: 0.3734783000416226
Epoch 29/50, Loss: 0.3768281936645508
Epoch 30/50, Loss: 0.37319568130705094
Epoch 31/50, Loss: 0.3560303780767653
Epoch 32/50, Loss: 0.3446658220556047
Epoch 33/50, Loss: 0.3398974703417884
Epoch 34/50, Loss: 0.340749376349979
Epoch 35/50, Loss: 0.32340293294853634
Epoch 36/50, Loss: 0.29859186377790237
Epoch 37/50, Loss: 0.2958988199631373
Epoch 38/50, Loss: 0.27048202024565804
Epoch 39/50, Loss: 0.258365273475647
Epoch 40/50, Loss: 0.21922836038801405
Epoch 41/50, Loss: 0.20390544169478947
Epoch 42/50, Loss: 0.1647645193669531
Epoch 43/50, Loss: 0.14581284837590325
Epoch 44/50, Loss: 0.13826308109694058
Epoch 45/50, Loss: 0.11470513252748384
Epoch 46/50, Loss: 0.09886708648668395
Epoch 47/50, Loss: 0.08508327189419004
Epoch 48/50, Loss: 0.08327979987694158
Epoch 49/50, Loss: 0.07419805746111605
Epoch 50/50, Loss: 0.065288371923897
Test Accuracy: 91.67%
```

Prediction of emotions of sample lines

Code :

```
# Sample inputs
sample_inputs = [
    "I love the new feature!",
    "I am satisfied with the customer service.",
    "This movie is not good!",
    "This software have lots of features.",
    "Incredible customer experience!",
    "I am anxious"
]

# Predict emotions for each sample input
for sample_input in sample_inputs:
    # Preprocess the sample input
    sample_input_indices = [word_to_idx[word] for word in
sample_input.split() if word in word_to_idx]

    # Check if the sample input is empty after filtering
    if not sample_input_indices:
        print(f"No valid words found in the sample input: {sample_input}")
    else:
        # Convert to tensor and add batch dimension
        sample_input_tensor =
torch.LongTensor(sample_input_indices).unsqueeze(0).to(device)

        # Pass through the trained model
        model.eval()
        with torch.no_grad():
            output = model(sample_input_tensor)

        # Interpret the model's output
        predicted_class = torch.argmax(output, dim=1).item()

        # Map the predicted class to the corresponding emotion label
        emotion_labels = {0: "Negative", 1: "Positive"}
        predicted_emotion = emotion_labels.get(predicted_class, "Unknown")

        print(f"The predicted emotion for the sample input
'{sample_input}' is: {predicted_emotion}")
```

```
The predicted emotion for the sample input 'I love the new feature!' is:
Positive
The predicted emotion for the sample input 'I am satisfied with the
customer service.' is: Positive
The predicted emotion for the sample input 'This movie is not good!' is:
Negative
The predicted emotion for the sample input 'This software have lots of
features.' is: Positive
The predicted emotion for the sample input 'Incredible customer
experience!' is: Positive
The predicted emotion for the sample input 'I am anxious' is: Unknown
```

Conclusion for Model 1

The model architecture is a simple neural network with an embedding layer, LSTM layer, and a linear layer for classification. The training process is carried out for 50 epochs, and the model achieves a test accuracy of **91.67%**. The model learned to understand the sentiment expressed in text and it correctly predicts positive sentiment for phrases like "I love the new feature!" and "Incredible customer experience!" It also identifies negative sentiment in phrases like "This movie is not good!" However, it struggles with ambiguous or less clear cases, as seen in the phrase "I am anxious," where it couldn't confidently predict an emotion.

Model 2

Changes were made to step5 of model 1 where we added an additional batch normalization layer between the LSTM and dropout layers. The code implements a learning rate scheduler (ReduceLROnPlateau) to adjust the learning rate based on training loss.

Code

```
class EmotionAnalysisModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(EmotionAnalysisModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.batch_norm = nn.BatchNorm1d(hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        lstm_out = lstm_out[:, -1, :]
        lstm_out = self.batch_norm(lstm_out)  # Apply batch normalization
        lstm_out = self.dropout(lstm_out)  # Apply dropout
        out = self.fc(lstm_out)
        return out
```

Test Accuracy of The Model

```
Epoch 1/50, Loss: 5.212891896565755
Epoch 2/50, Loss: 4.99224673377143
Epoch 3/50, Loss: 4.902364730834961
Epoch 4/50, Loss: 4.60579087999132
Epoch 5/50, Loss: 4.311837037404378
Epoch 6/50, Loss: 3.9687681198120117
Epoch 7/50, Loss: 3.5991027620103626
Epoch 8/50, Loss: 3.12363330523173
Epoch 9/50, Loss: 2.501327223247952
Epoch 10/50, Loss: 2.0091420544518366
Epoch 11/50, Loss: 1.4027652210659451
Epoch 12/50, Loss: 1.1353367567062378
Epoch 13/50, Loss: 0.675278902053833
Epoch 14/50, Loss: 0.5552616500192218
Epoch 15/50, Loss: 0.3590041597684224
Epoch 16/50, Loss: 0.19627143401238653
Epoch 17/50, Loss: 0.13862000323004192
Epoch 18/50, Loss: 0.12142143522699674
Epoch 19/50, Loss: 0.08268820266756746
Epoch 20/50, Loss: 0.059918465092778206
Epoch 21/50, Loss: 0.04743958595726225
Epoch 22/50, Loss: 0.09386506314492887
Epoch 23/50, Loss: 0.04212546803885036
Epoch 24/50, Loss: 0.034734474081132144
Epoch 25/50, Loss: 0.04235730232256982
Epoch 26/50, Loss: 0.023594688727623887
Epoch 27/50, Loss: 0.02539259402288331
Epoch 28/50, Loss: 0.01916065610324343
Epoch 29/50, Loss: 0.03082093187711305
Epoch 30/50, Loss: 0.025313820224255323
Epoch 31/50, Loss: 0.018674598230669897
Epoch 32/50, Loss: 0.020553952341692314
Epoch 33/50, Loss: 0.02424334010316266
Epoch 34/50, Loss: 0.017769804845253628
Epoch 35/50, Loss: 0.015171972698428564
Epoch 36/50, Loss: 0.03093576767585344
Epoch 37/50, Loss: 0.08671659797740479
Epoch 38/50, Loss: 0.051407156706166766
Epoch 39/50, Loss: 0.014900382944486208
Epoch 40/50, Loss: 0.03975348195268048
Epoch 41/50, Loss: 0.024840969204281766
Epoch 42/50, Loss: 0.0878054055178331
Epoch 43/50, Loss: 0.10371428821235895
Epoch 44/50, Loss: 0.02248115252910389
Epoch 45/50, Loss: 0.013741839909926057
Epoch 46/50, Loss: 0.010360020186959041
Epoch 47/50, Loss: 0.013167030695411894
Epoch 48/50, Loss: 0.009693657785343627
Epoch 49/50, Loss: 0.006986469418431322
Epoch 50/50, Loss: 0.008784590871073306
Test Accuracy: 95.83%
```

Prediction of emotions of sample lines

```
The predicted emotion for the sample input 'I love the new feature!' is:
Positive
The predicted emotion for the sample input 'I am satisfied with the
customer service.' is: Positive
The predicted emotion for the sample input 'This movie is not good!' is:
Negative
The predicted emotion for the sample input 'This software have lots of
features.' is: Negative
The predicted emotion for the sample input 'Incredible customer
experience!' is: Negative
The predicted emotion for the sample input 'I am anxious' is: Positive
```

Conclusion for Model 2

The model underwent a training regimen spanning 50 epochs, culminating in a test accuracy of **95.83%**. While Model 2 demonstrated superior accuracy compared to Model 1, it was not without its missteps in predictions. Notably, it erroneously classified positive statements like “`This software have lots of features.`” and “`Incredible customer experience!`” as negative. Conversely, it incorrectly identified the negative sentiment in “`I am anxious`” as positive. Despite Model 2's higher accuracy, these instances underscore its limitations in accurately discerning the sentiment of certain samples.

Model 3

Changes were made to step5 of model 1 where pretrained_embeddings are provided, the weights of the embedding layer are initialized with these pre-trained embeddings. This is useful when leveraging pre-trained word embeddings, such as Word2Vec or GloVe. The weights are set to be non-trainable (requires_grad = False) in case of pre-trained embeddings.

Code:

```
class ImprovedEmotionAnalysisModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim,
pretrained_embeddings=None):
        super(ImprovedEmotionAnalysisModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        if pretrained_embeddings is not None:

self.embedding.weight.data.copy_(torch.from_numpy(pretrained_embeddings))
            self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True,
bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        lstm_out = lstm_out[:, -1, :]
        lstm_out = self.dropout(lstm_out)
        out = self.fc(lstm_out)
        return out
```

Test Accuracy of The Model

```
Epoch 1/50, Loss: 4.298942142062717
Epoch 2/50, Loss: 2.4066073894500732
Epoch 3/50, Loss: 0.8778071800867716
Epoch 4/50, Loss: 0.5014093981848823
Epoch 5/50, Loss: 0.4190385606553819
Epoch 6/50, Loss: 0.4098073012299008
Epoch 7/50, Loss: 0.39207568764686584
Epoch 8/50, Loss: 0.4053659869564904
Epoch 9/50, Loss: 0.38669821951124406
Epoch 10/50, Loss: 0.38769100109736127
Epoch 11/50, Loss: 0.39474110470877755
Epoch 12/50, Loss: 0.39178109500143266
Epoch 13/50, Loss: 0.3780011369122399
Epoch 14/50, Loss: 0.384607646200392
Epoch 15/50, Loss: 0.34878357582622105
Epoch 16/50, Loss: 0.33234446081850266
Epoch 17/50, Loss: 0.3111840420299106
Epoch 18/50, Loss: 0.2962263855669234
Epoch 19/50, Loss: 0.2789573238955604
Epoch 20/50, Loss: 0.23815032177501255
Epoch 21/50, Loss: 0.21328215135468376
Epoch 22/50, Loss: 0.1740485429763794
Epoch 23/50, Loss: 0.15266337659623888
Epoch 24/50, Loss: 0.13207165648539862
Epoch 25/50, Loss: 0.11312453986869918
Epoch 26/50, Loss: 0.11571580585506228
Epoch 27/50, Loss: 0.09975932165980339
Epoch 28/50, Loss: 0.09674576297402382
Epoch 29/50, Loss: 0.08749340826438533
Epoch 30/50, Loss: 0.08410173861516847
Epoch 31/50, Loss: 0.08670817501842976
Epoch 32/50, Loss: 0.07847022472156419
Epoch 33/50, Loss: 0.07839789179464181
Epoch 34/50, Loss: 0.08933944130937259
Epoch 35/50, Loss: 0.07552709989249706
Epoch 36/50, Loss: 0.0832259746061431
Epoch 37/50, Loss: 0.08482471501661672
Epoch 38/50, Loss: 0.077828049659729
Epoch 39/50, Loss: 0.08589657851391369
Epoch 40/50, Loss: 0.06929195848190123
Epoch 41/50, Loss: 0.07019840718971358
Epoch 42/50, Loss: 0.07724801844192876
Epoch 43/50, Loss: 0.04984689628084501
Epoch 44/50, Loss: 0.04687118778626124
Epoch 45/50, Loss: 0.03701224115987619
Epoch 46/50, Loss: 0.02200316648102469
Epoch 47/50, Loss: 0.02892321317146222
Epoch 48/50, Loss: 0.025212829052988026
Epoch 49/50, Loss: 0.02620111669724186
Epoch 50/50, Loss: 0.029320076832340822
Test Accuracy: 93.75%
```

Prediction of emotions of sample lines

```
The predicted emotion for the sample input 'I love the new feature!' is:
Positive
The predicted emotion for the sample input 'I am satisfied with the
customer service.' is: Positive
The predicted emotion for the sample input 'This movie is not good!' is:
Positive
The predicted emotion for the sample input 'This software have lots of
features.' is: Negative
The predicted emotion for the sample input 'Incredible customer
experience!' is: Positive
The predicted emotion for the sample input 'I am anxious' is: Negative
```

Conclusion for Model 3

After completing a training cycle of 50 epochs, the model achieved a test accuracy of **93.75%**. Model 3 showed an improvement over Model 1 but fell short of Model 2's performance. However, its predictive capabilities were not flawless. For instance, it mistakenly interpreted the negative statement “This movie is not good!” as positive. Similarly, it misjudged the positive sentiment in “This software has lots of features.” as negative. Although Model 3 outperformed Model 1 in terms of accuracy, these examples highlight its challenges in consistently and accurately evaluating the sentiment in specific statements.

Conclusion

In our project, we tested three different types of neural networks to see how well they can figure out if a sentence is positive or negative. Each model was a bit different in how it was built and trained.

Model 1 was pretty basic. It used an embedding layer, an LSTM layer, and a linear layer for figuring out sentiments. After training it for 50 epochs, it got 91.67% right on our test. It was good at picking up obvious happy or sad sentences, but got confused with trickier ones like "I am anxious."

Then we made Model 2, which was a bit more complex than Model 1. We added a batch normalization layer and used a learning rate scheduler to improve training. This model did better, scoring 95.83% on the test, but it still made some mistakes, like mixing up positive and negative sentiments in certain sentences.

Finally, we created Model 3. This one used pre-trained embeddings, which are like shortcuts to help the model understand words better. Model 3 did better than Model 1 with a score of

93.75%, but it wasn't as good as Model 2. Like the others, it sometimes got confused, especially with sentences that weren't straightforward.

In short, each model had its strengths and weaknesses. They all got better at telling if a sentence was happy or sad, but none of them were perfect, especially when the sentences were a bit vague or complicated. This shows that figuring out emotions in text is a tough job and we still have more work to do to make these models better.

OpenAI API

As suggested by the professor, we tried the openAI API to predict the sentiments of the sample data. The language model used here was text-davinci-002. Following is the code to call the API followed by the output.

Steps to implement:

To connect to the OpenAI and call the API, we need to follow below steps:

1. **Get an OpenAI Account:** First off, we need to sign up with OpenAI. This gets us access to their API and the all-important API keys.
2. **Install OpenAI Package:** Next, we need to get the OpenAI package onto our Python setup. It's easy - just run pip install openai in our Python environment, like in a Colab notebook.
3. **Import OpenAI in Our Code:** To use OpenAI's tools, we have to include them in our code. This is as simple as adding import openai at the start of our script.
4. **Set Up OpenAI Keys:** Now, we need to tell our code about our OpenAI keys. These keys are like secret codes that let us use the API. We do this by adding `openai.organization = "YOUR-ORGANIZATION-KEY"` and `openai.api_key = "YOUR-API-KEY"` in our code, swapping in the actual keys from OpenAI account.
5. **Create the Prompt:** This part's fun - we create a prompt. It's basically giving OpenAI instructions on what we want. We can get creative here and use Prompt Engineering, which is all about crafting the right instructions to get the results we want.
6. **Make the API Request:** Finally, we ask the API to do its magic with `openai.Completion.create`. We pass it our prompt and any other details we want, and it gives us back a response. We can see what it comes up with by checking out the `response["choices"][0]["text"]` part of the response.

Code

```
!pip install openai==0.28
```

```python
import openai
```

```python
def setup_openai_api(api_key):
  openai.api_key = api_key
```

```python
# Function to classify the sentiment of the given text using OpenAI's API.
def classify_sentiment(text, api_key):
    #print(f"The following text: '{text}' expresses a
[positive/negative/unknown] sentiment.")
    setup_openai_api(api_key)
    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=f"The following text: '{text}' expresses a
[positive/negative/unknown] sentiment.",
        max_tokens=60
    )
    return response.choices[0].text.strip()
```

```python
def main():
    # OpenAI API key
    api_key = 'replace with your actual API key'

    # Sample inputs for testing
    sample_inputs = [
        "I love the new feature!",
        "I am satisfied with the customer service.",
        "This movie is not good!",
        "This software have lots of features.",
        "Incredible customer experience!",
        "I am anxious"
    ]

    # Classify sentiment for each sample input
    for text in sample_inputs:
        sentiment = classify_sentiment(text, api_key)
        print(f"'{text}' is: {sentiment.capitalize()}")
```

```python
main()
```

Output

```
'I love the new feature!' is: Positive
'I am satisfied with the customer service.' is: The sentiment in the text
is positive.
'This movie is not good!' is: The sentiment in the text is negative.
'This software have lots of features.' is: Negative
'Incredible customer experience!' is: Positive
'I am anxious' is: The sentiment expressed in the text is negative.
```

References

https://community.openai.com/t/help-getting-access-to-text-davinci-003-in-api/146178
https://medium.com/@yosik81/code-crafting-from-input-forms-an-interface-for-function-creation-with-openai-api-bd658f7d5b4b