Project 2 - Emotion

Henrik Daniel Christensen[hench13@student.sdu.dk]
Frode Engtoft Johansen[fjoha21@student.sdu.dk]

DM873: Deep Learning
University of Southern Denmark, SDU
Department of Mathematics and Computer Science

1 Introduction

The aim of this project is to design and implement two deep learning models for text/sentiment classification, capable of identifying the emotion conveyed in a given text. The model is trained on the *dair-ai/emotion* dataset, a widely used collection of 20,000 labeled tweets available through Hugging Face. This dataset is divided into six emotion categories: sadness (0), joy (1), love (2), anger (3), fear (4), and surprise (5). Of the total dataset, 2,000 tweets are reserved exclusively for final testing.

2 Label Distribution

We started by analyzing the distribution of the labels in the dataset, see Figure 1.

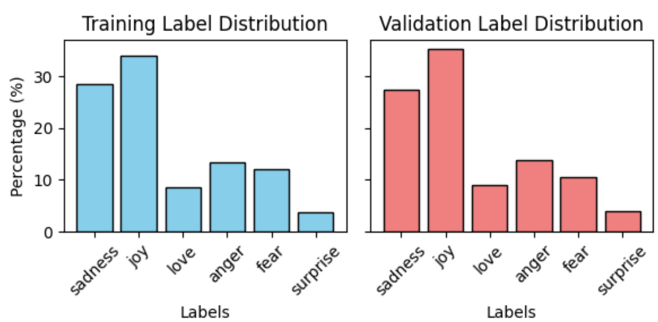


Fig. 1. Distribution of the labels in the dataset.

From the figure, it is evident that the dataset is highly imbalanced, with the majority of tweets labeled as either joy or sadness. However, the validation set exhibits a roughly similar distribution of labels, suggesting that the test set is likely to follow a comparable pattern.

This imbalance in the training data is expected to impact the model's performance, particularly for less represented labels, as sentences labeled with joy are less likely to be classified correctly. This challenge must be carefully considered when evaluating the model's performance and interpreting the results.

3 Preprocessing

To enable a machine learning model to process textual data effectively, it is essential to convert the text into numerical representations. The first step in this process is tokenizing the text, which involves splitting sentences into smaller units, such as words. For this task, we utilize a custom RegexpTokenizer from the NLTK library, which filters the text to retain only words, numbers, and a limited set of special characters. This ensures that irrelevant symbols are excluded while preserving the meaningful components of the text.

Following tokenization, the text is further refined by removing stopwords and applying stemming. Stopwords, which are frequently used words such as "the," "is," and "and," are removed as they are unlikely to contribute significant value to sentiment analysis. This is particularly important when using a limited sequence length for input sentences, as it allows the model to focus on more informative features while reducing the size of the vocabulary. Consequently, the model learns fewer parameters, which can improve its generalization and efficiency.

In addition, we apply stemming using the PorterStemmer from the NLTK library, which reduces words to their base or root forms (e.g., "feeling" becomes "feel"). This process further reduces the vocabulary size by grouping inflected or derived word forms together, helping the model to recognize similar meanings and improve its performance.

To better understand the text data after preprocessing, we visualize the most common words using a WordCloud, as shown in Figure 2. This provides an intuitive way to analyze the prominent terms in the dataset and assess whether the preprocessing aligns with the goals of the sentiment analysis task.

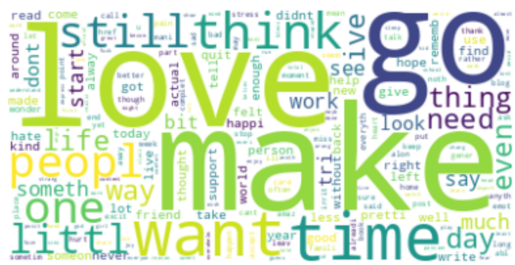


Fig. 2. WordCloud of the vocabulary.

As expected many emotional words are present in the vocabulary, e.g. 'love', 'friend', 'hate', 'never', 'go', etc.

To be able to process the text data, we need to choose a maximum sentence length. A boxplot of the distribution of the lengths of the sequences can be seen in Figure 3.

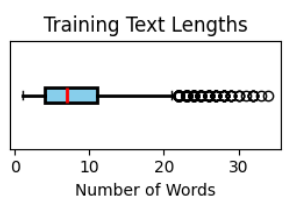


Fig. 3. Distribution of the lengths of the sequences.

From the boxplot, we see that the majority of the sequences have a length of around 8 words. We choose to set the maximum sequence length to 10 words. This means that all sequences longer than 10 words are truncated, and all sequences shorter than 23 words are padded with a special token, '<PAD>', to make them all the same length. In addition, we discovered that some sentences are very short, therefore we choose to remove all sentences with a length less than 3 words.

Having our cleaned and tokenized text data, we build a vocabulary from the training dataset. The vocabulary is built by mapping each unique word to an integer. The vocabulary for the training dataset ended up being 10,336 words.

Then, we convert the text data to integers by mapping each word in the text data to the corresponding integer in the vocabulary and pad the sequences to the maximum sequence length. This is done for both the training, validation and test datasets. The text data is now ready to be used for a model.

4 Models

Two different models were developed for the sentiment analysis task. The first model is a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) cells, and the second model is a Transformer model.

4.1 Model 1: RNN-LSTM

Model Architecture We chose to begin with a Recurrent Neural Network (RNN) model using LSTM (Long Short-Term Memory) cells due to their proven effectiveness in capturing long-term dependencies in sequential data, such as text. LSTMs are particularly suited for this task because they address the vanishing gradient problem often encountered in standard RNNs.

The architecture of the model was designed to balance complexity and performance, resulting in a total of 1,117,734 trainable parameters. Key considerations behind the design are as follows:

- Embedding Layer: The vocabulary size of 10,336 and embedding dimension of 75 were chosen to compactly represent the input text while capturing semantic relationships between words effectively. This dimensionality ensures the embeddings are rich enough for the sentiment classification task without being computationally excessive.
- LSTM Layer: A single LSTM layer with 256 hidden units was selected to provide sufficient capacity to capture sequential patterns in the text. This configuration was chosen to avoid overfitting while retaining the ability to model complex dependencies in the data.
- Dropout Layer: A dropout rate of 50% was applied to the LSTM outputs to reduce the risk of overfitting by preventing the model from becoming overly reliant on specific neurons. This regularization technique ensures better generalization to unseen data.
- Output Layer: The final linear layer maps the 256 features to the 6 output classes, corresponding to the emotions in the dataset.

The specific parameters mentioned above was found using a combination of empirical testing (grid search) and theoretical considerations to achieve a balance between model complexity and performance.

Training and Evaluation The model training was conducted using the AdamW optimizer with a learning rate of 0.001 and a batch size of 16. AdamW was chosen due to its effectiveness in handling sparse gradients and its built-in weight decay mechanism, which helps prevent overfitting. The batch size of 16 strikes a balance between computational efficiency and the stability of gradient updates.

For the loss function, we used CrossEntropyLoss, which is well-suited for multi-class classification problems like this one, where the task involves predicting one of six emotion categories. To further regularize the model and reduce the risk of overfitting, an L2-norm penalty (weight decay) of 0.0001 was applied to the weights, encouraging smaller parameter values and promoting a simpler model. The model was trained for 4 epochs, as using more epochs lead to overfitting. The final test accuracy and F1-score is summarized in Table 2.

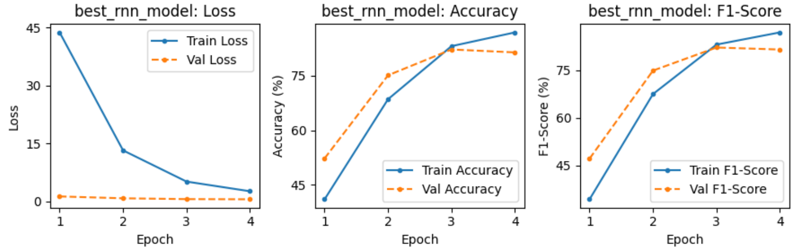


Fig. 4. Training and validation loss, accuracy, and F1-score for the RNN-LSTM model.

Label	Precision	Recall	F1-Score
sadness	0.91	0.86	0.89
joy	0.85	0.86	0.85
love	0.62	0.69	0.65
anger	0.84	0.81	0.82
fear	0.81	0.85	0.83
surprise	0.65	0.65	0.65
accuracy			0.83
weighted avg	0.84	0.83	0.83

Table 1. RNN-LSTM model performance on test set.

From the above table, we can see that the model performs well on sadness, joy, anger and fear, but not as well on love and surprise. This is likely due to the imbalanced distribution of the classes in the dataset as discussed in Section 2, see Figure 1.

4.2 Model 2: Transformer

The model is made from the solution to exercise sheet 5. Very few elements have been changed since it follows the pages in the Bishop book closely. The only big changes that has been made to the model itself is added dropout layers and positional encoding. We also experimented with randomly removing 2 tokens from each sentence to increase generalisation but it didn't work. WE also tried adding 2 more linear layers to the transformer block, but this was detrimental to the accuracy so we removed them again. We chose to use a transformer model as it seems like a good architecture for working with text since it makes words weigh differently depending on which other words are present in the sentence, and with positional encoding also where they are in that sentence. This is important since the meaning of words change depending on other words, and position in sentence. We used sinusoidal position embedding since this is the positional embedding used in the Bishop book. The L in the sinusoidal position embedding is 10000. The example in the Bishop book had an L of 30, and some other examples we encountered had an L of 10000. We chose 10000 as the value for L since we encountered it multiple places.

We used cross entropy loss as a our loss function since this is a good loss function for classification tasks. This is the number of heads in the multi head self attention mechanism. d_model should preferably be divisible by this number so each head computes the same size of input. We chose 8 heads since having too many heads can be inefficient and since we have d_model of 128, choosing more heads would lead to quite few dimensions per head, which would make it hard for them to capture any useful features. The mlp factor controls the amount of nodes in the linear layers of the transformer block. We use an mlp factor of 4. This was the default value and decreasing this value hit the performance of the model. Increasing the mlp factor would increase the training time so we decided to not do that. d_model is the dimensions of each token. We have a vocabulary of 10336 so we made this pretty big to ensure that each word is able to be uniquely represented.

We have 6 layers of transformer blocks. According to the slides, 6 layers is a normal amount of layers and 12 layers is for very large models. We experimented with 6 and 12 layers and found only a small performance increase with 12 layers, but a significant increase in training time, so we decided that we would rather train quicker to test values for other parameters. When training the transformer models with different parameters we found out that they would quickly overfit, so we needed to generalise the model. To do this we added two dropout layers on the linear layers of the transformer blocks. The dropout chance for the first layer is 50% and for the second layer is 25%. We tested with both higher and lower dropout chances and this worked the best. We chose adam as the optimizer. From the last project our and other peoples takeaway were that adam was a good optimizer in general, and since were not using weight decay, we don't need to use adamW. We used a learning rate of 0,0003. This was discussed as a good learning rate from the last project, and our experimenting also led us to this learning rate. We experimented a bit with using weight decay but it did not work well so we decided not to use it in our final model. When testing different parameters, we used 50 epochs. The transformer models quickly overfitted so we didn't need more epochs. We experimented with both L1 and L2 regularisation. L1 regularisation worked the best, and that is probably because it makes some weights go to 0, which indicates those words aren't important. In sentences, some words tells more than other words, so it makes sense that this would be good.

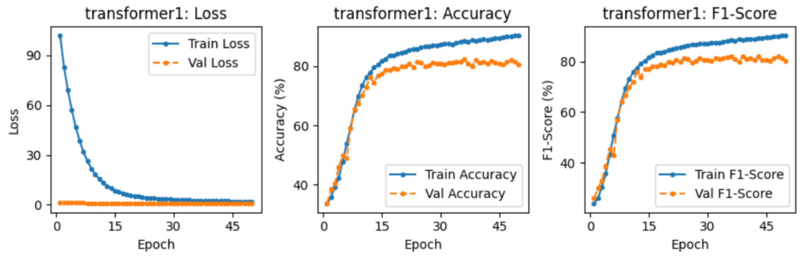


Fig. 5. Training and validation loss, accuracy, and F1-score for the transformer model.

Label	Precision	Recall	F1-Score
sadness	0.89	0.82	0.85
joy	0.81	0.88	0.85
love	0.68	0.59	0.63
anger	0.77	0.81	0.79
fear	0.80	0.80	0.80
surprise	0.67	0.56	0.61
accuracy			0.81
weighted avg	0.81	0.81	0.81

Table 2. Transformer model performance on test set.

Despite a lot of parameter tuning, we could not get the transformer model to get a better performance. We even tried adding extra linear layers in the transformer block, and removing random words from the sentences for better generalisation.

5 Analysis and Final Prediction

Of the two models, the performance of the recurrent neural network were better with an accuracy of 83% to the transformer models accuracy of 81%. Both architectures are good at predicting based on text, so its no big surprise that they are close in performance.

Since the transformer model is the worst performing, we decided to take a closer look at some failed classification cases to see why it performed as it did. These are the 7 sentences we looked at.

Number	Sentence	Transformer prediction	Actual classification
1	im updating my blog because i feel shitty	Joy	Sadness
2	i never make her separate from me because i don t ever want her to feel like i m ashamed with her	Joy	Sadness
3	i left with my bouquet of red and yellow tulips under my arm feeling slightly more optimistic than when i arrived	Sadness	Joy
4	i cant walk into a shop anywhere where i do not feel uncomfortable	Joy	Fear
5	i explain why i clung to a relationship with a boy who was in many ways immature and uncommitted despite the excitement i should have been feeling for getting accepted into the masters program at the university of virginia	Anger	Joy
6	i jest i feel grumpy tired and pre menstrual which i probably am but then again its only been a week and im about as fit as a walrus on vacation for the summer	Joy	Anger
7	i find myself in the odd position of feeling supportive of	Anger	Love

Table 3. Sentence Classifications with Transformer prediction and Actual classification

Looking at these sentences my intuition would be that the model looks at keywords to decide which category it belongs to. Some of the sentences lack any keywords that indicate what feeling they are associated with. The first sentence has a keyword 'shitty', but it might be a rare word in the training set. Sentence 1 also contains the word updating which is probably a positive word for the model, which is why it predicts joy. Sentence 6 contain the word 'vacation' which is probably why the model predicts joy for that sentence. This could indicate it is looking for keywords. If you were looking to deliberately make the model fail, we predict that missspellings, irony and figurative language would be some good techniques to do so.

To confirm our suspicions we tried to edit some of the sentences to get them correctly classified. Changing sentence 1 to "im updating my blog because i feel sad" changed the classification to sadness which is the correct one. This indicates that keywords are important. On the other hand, changing sentence 5 to "i explain why i clung to a relationship with a boy who was in many ways despite the excitement i should have been feeling for getting accepted into the masters program at the university of virginia" did not change the prediction or even the confidence which is a big surprise since the only change in that sentence is removing "immature and uncommitted" which seems like the most negative words in that sentence, so maybe the model is less keyword focused than anticipated.

6 Pretrained Model (DistilBERT)

To evaluate how our model compares to a larger pretrained model, we utilized the DistilBERT base model (uncased) from Hugging Face. DistilBERT is a distilled version of BERT, pretrained on a large corpus comprising 11,038 books and the English Wikipedia. It features a vocabulary size of 30,000 tokens and contains approximately 67 million parameters, making it significantly larger and more complex than our models. To adapt DistilBERT to our sentiment analysis task, we fine-tuned the model for 5 epochs using a small initial learning rate of 5×10^{-6}. This low learning rate ensures stable and effective fine-tuning of the pretrained weights without overfitting to our dataset. The performance of DistilBERT on our task is summarized in Table 4.

Label	Precision	Recall	F1-Score
sadness	0.92	0.93	0.92
joy	0.90	0.92	0.91
love	0.74	0.71	0.72
anger	0.90	0.89	0.89
fear	0.87	0.89	0.88
surprise	0.82	0.61	0.70
accuracy			0.89
weighted avg	0.89	0.89	0.89

Table 4. DistilBERT model performance on test set.

From the above results, we see that the pretrained model performs generally better than our two models. Notice that the performance on the less presents classes love and surprise is also lower as seen in the previous models.

7 Conclusion

We constructed two different models to complete the task, a recurrent neural network with LSTM and 256 hidden units, and a transformer model with 6 transformer layers. The recurrent neural network had the best test accuracy of 83%, while the transformer model had a test accuracy of 81%. We tried using a pretrained model distilBERT on our task to see how it would fare. It got a test accuracy of 89%. As expected our models were worse. This can be explained by the fact that distilBERT has more parameters and a much bigger training set than our models.

Overall we are satisfied with our result but it could have been improved. Initially an accuracy of 83% seems quite bad, but even the pretrained model had a test accuracy of less than 90% so compared to that, it does not seem too bad.

7.1 Individual Contributions

	Henrik Daniel Christensen	**Frode Engtoft Johansen**
Code	Task 1, 2, 5	Task 3, 4
Report	Section 1, 2, 3, 4.1, 6	Section 4.2, 5, 7

Table 5. Individual contributions.

A Code

Python Files

File: emotion_dataset.py

```
import torch
from torch.utils.data import Dataset

class EmotionDataset(Dataset):
   def __init__(self, data, labels):
       self.data = data
       self.labels = labels

   def __len__(self):
       return len(self.data) # return number of samples

   def __getitem__(self, idx):
       return torch.LongTensor(self.data[idx]), torch.tensor(self.labels[idx], dtype=torch.long) # return sample and l
```

File: loader.py

```
import os
import pandas as pd
from datasets import load_dataset

def loader(train_csv_path, val_csv_path, test_csv_path):
   # Check if the files exist; if not, load from the remote source
   if not (os.path.exists(train_csv_path) and os.path.exists(val_csv_path) and os.path.exists(test_csv_path)):
       print("Data files not found. Loading dataset from remote source...")

       os.makedirs("data", exist_ok=True)

       # Load the dataset from Hugging Face
       ds = load_dataset("dair-ai/emotion", "split")

       label_names = ds["train"].features["label"].names

       # Save train data
       train_data = {
           "text": ds["train"]["text"],
           "label": ds["train"]["label"],
           # Convert label indices to label names
           "label_name": [label_names[label] for label in ds["train"]["label"]]
       }
       pd.DataFrame(train_data).to_csv(train_csv_path, index=True)

       # Save validation data
       val_data = {
           "text": ds["validation"]["text"],
           "label": ds["validation"]["label"],
           "label_name": [label_names[label] for label in ds["validation"]["label"]]
       }
       pd.DataFrame(val_data).to_csv(val_csv_path, index=True)

       # Save test data
       test_data = {
           "text": ds["test"]["text"],
           "label": ds["test"]["label"],
           "label_name": [label_names[label] for label in ds["test"]["label"]]
       }
       pd.DataFrame(test_data).to_csv(test_csv_path, index=True)

   train_df = pd.read_csv(train_csv_path, index_col=0)
   val_df = pd.read_csv(val_csv_path, index_col=0)
   test_df = pd.read_csv(test_csv_path, index_col=0)

   return train_df, val_df, test_df
```

File: metrics.py

```
import os
from sklearn.metrics import (
   accuracy_score,
   f1_score,
```

```
    precision_score,
    recall_score,
    classification_report,
    confusion_matrix,
)

def compute_metrics(true_labels, predicted_labels, labels):
    accuracy = round(accuracy_score(true_labels, predicted_labels), 4)
    f1 = round(f1_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    precision = round(precision_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    recall = round(recall_score(true_labels, predicted_labels, average="weighted", zero_division=0), 4)
    class_report = classification_report(true_labels, predicted_labels, target_names=labels, zero_division=0)
    conf_matrix = confusion_matrix(true_labels, predicted_labels)
    return {
        "accuracy": accuracy,
        "f1": f1,
        "precision": precision,
        "recall": recall,
        "class_report": class_report,
        "conf_matrix": conf_matrix.tolist()
    }

def print_metrics(metrics):
    for key, value in metrics.items():
        if key == "conf_matrix":
            print("confusion matrix:")
            for row in value:
                print(row)
            print()
        elif key == "class_report":
            print("classification report:")
            print(value)
        else:
            print(f"{key}: {value}")

def save_metrics(label, metrics):
    os.makedirs("results", exist_ok=True)
    with open(f"results/{label}_metrics.txt", "w") as f:
        f.write(f"Accuracy Score: {metrics['accuracy']}\n")
        f.write(f"F1-Score: {metrics['f1']}\n")
        f.write(f"Precision: {metrics['precision']}\n")
        f.write(f"Recall: {metrics['recall']}\n\n")
        f.write("Classification Report:\n")
        f.write(metrics['class_report'] + "\n")
        f.write("Confusion Matrix:\n")
        for row in metrics['conf_matrix']:
            f.write(str(row) + "\n")
```

File: plot_scores.py

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

def plot_scores(results, label):
    _, axes = plt.subplots(1, 3, figsize=(9, 3))  # Adjust the layout for 3 subplots

    epochs = range(1, results["num_epochs"] + 1)

    # Plot loss
    axes[0].plot(epochs, results["train_losses"], marker="o", markersize=3, label="Train Loss")
    axes[0].plot(epochs, results["val_losses"], marker="o", markersize=3, linestyle="--", label="Val Loss")
    axes[0].set_title(f"{label}: Loss")
    axes[0].set_xlabel("Epoch")
    axes[0].set_ylabel("Loss")
    axes[0].legend(loc="best")

    # Convert accuracy and F1-score to percentages
    train_accuracies = [acc * 100 for acc in results["train_accuracies"]]
    val_accuracies = [acc * 100 for acc in results["val_accuracies"]]
    train_f1_scores = [f1 * 100 for f1 in results["train_f1_scores"]]
    val_f1_scores = [f1 * 100 for f1 in results["val_f1_scores"]]

    # Plot accuracy
    axes[1].plot(epochs, train_accuracies, marker="o", markersize=3, label="Train Accuracy")
    axes[1].plot(epochs, val_accuracies, marker="o", markersize=3, linestyle="--", label="Val Accuracy")
    axes[1].set_title(f"{label}: Accuracy")
```

```
    axes[1].set_xlabel("Epoch")
    axes[1].set_ylabel("Accuracy (%)")
    axes[1].legend(loc="best")

    # Plot F1-score
    axes[2].plot(epochs, train_f1_scores, marker="o", markersize=3, label="Train F1-Score")
    axes[2].plot(epochs, val_f1_scores, marker="o", markersize=3, linestyle="--", label="Val F1-Score")
    axes[2].set_title(f"{label}: F1-Score")
    axes[2].set_xlabel("Epoch")
    axes[2].set_ylabel("F1-Score (%)")
    axes[2].legend(loc="best")

    # Set the number of ticks on the x and y axes for all plots
    for ax in axes:
        ax.yaxis.set_major_locator(MaxNLocator(nbins=4))
        ax.xaxis.set_major_locator(MaxNLocator(nbins=4))

    # Adjust layout and display the plots
    plt.tight_layout()
    plt.show()
```

File: predict.py

```
import os
import torch
import pandas as pd
from metrics import compute_metrics, print_metrics, save_metrics

def decode_tokens(token_ids, reverse_vocab):
    words = [reverse_vocab[token] for token in token_ids if token in reverse_vocab]
    return " ".join(words)

def predict(label, model, device, loader, label_map, reverse_vocab):
    model.to(device)
    model.eval()

    # Store predictions
    predictions = []
    true_labels = []
    predicted_labels = []

    # Ensure the results directory exists
    os.makedirs("results", exist_ok=True)

    with torch.no_grad():
        for batch in loader:
            sentences, labels = batch
            labels = labels.to(device)

            # Forward pass
            outputs = model(sentences.to(device))
            probabilities = torch.softmax(outputs, dim=1)
            predicted = torch.argmax(probabilities, dim=1)

            # Collect true and predicted labels for metrics
            true_labels.extend(labels.cpu().numpy())
            predicted_labels.extend(predicted.cpu().numpy())

            # Convert tokenized tensors back to text using decode_tokens()
            decoded_sentences = [
                decode_tokens(sentence.tolist(), reverse_vocab)
                for sentence in sentences
            ]

            for i, decoded_sentence in enumerate(decoded_sentences):
                true_label = label_map[labels[i].item()]
                pred_label = label_map[predicted[i].item()]
                confidence = probabilities[i][predicted[i].item()].item()
                correct = true_label == pred_label

                predictions.append({
                    "Sentence": decoded_sentence,
                    "Correct": correct,
                    "True Label": true_label,
                    "Predicted Label": pred_label,
                    "Confidence (%)": f"{confidence * 100:.2f}"
```

```
            })

    # Save predictions to a CSV file
    predictions_df = pd.DataFrame(predictions)
    predictions_file = f"results/{label}_predictions.csv"
    predictions_df.to_csv(predictions_file, index=False)

    # Compute metrics
    metrics = compute_metrics(true_labels, predicted_labels, label_map.values())
    print_metrics(metrics)
    save_metrics(label, metrics)
```

File: train_model.py

```
import os
import time
import numpy as np
from collections import Counter

from torch import nn
import torch
import torch.optim as optim
from torch.nn.utils import clip_grad_norm_
from torchinfo import summary

from sklearn.metrics import accuracy_score, f1_score

from metrics import compute_metrics, print_metrics, save_metrics

def log_undefined(predicted_labels, labels):
    counts = Counter(predicted_labels)
    for idx, label in enumerate(labels):
        if counts[idx] == 0:
            print(f"Warning: No predictions for label '{label}' (index {idx}).")


def log_undefined(predicted_labels, labels):
    """Logs warnings for labels that are not predicted."""
    counts = Counter(predicted_labels)
    for idx, label in enumerate(labels):
        if counts[idx] == 0:
            print(f"Warning: No predictions for label '{label}' (index {idx}).")


def train_model(
    label,
    model,
    train_loader,
    val_loader,
    label_map,
    device,
    optimizer_type="Adam",
    learning_rate=0.001,
    momentum=0.9,
    weight_decay=0.0,
    step_size=None,
    gamma=0.5,
    reg_type=None,
    reg_lambda=0.0,
    num_epochs=30,
    grad_clip=0.0,
):
    """Trains a PyTorch model and logs metrics for each epoch."""
    # Move the model to the device
    model = model.to(device)

    # Define the loss function
    criterion = nn.CrossEntropyLoss()

    # Select optimizer
    if optimizer_type == "SGD":
        optimizer = optim.SGD(
            model.parameters(), lr=learning_rate, momentum=momentum, weight_decay=weight_decay
        )
    elif optimizer_type == "Adam":
        optimizer = optim.Adam(
```

```
            model.parameters(), lr=learning_rate, weight_decay=weight_decay
        )
    elif optimizer_type == "AdamW":
        optimizer = optim.AdamW(
            model.parameters(), lr=learning_rate, weight_decay=weight_decay
        )
    else:
        raise ValueError(f"Unknown optimizer type: {optimizer_type}")

    # Learning rate scheduler
    scheduler = None
    if step_size is not None and gamma is not None:
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

    # Metrics storage
    train_losses, train_accuracies, train_f1_scores = [], [], []
    val_losses, val_accuracies, val_f1_scores = [], [], []

    total_start_time = time.time()

    for epoch in range(num_epochs):
        start_time = time.time()

        # Training phase
        model.train()

        epoch_total_train_loss = 0.0
        epoch_total_train_samples = 0
        epoch_train_true_labels = []
        epoch_train_predicted_labels = []

        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            # Apply regularization
            if reg_lambda > 0.0 and reg_type is not None:
                if reg_type == "L1":
                    l1_norm = sum(param.abs().sum() for param in model.parameters())
                    loss += reg_lambda * l1_norm
                elif reg_type == "L2":
                    l2_norm = sum(param.pow(2).sum() for param in model.parameters())
                    loss += reg_lambda * l2_norm

            loss.backward()

            if grad_clip > 0:
                clip_grad_norm_(model.parameters(), grad_clip)

            optimizer.step()

            epoch_total_train_loss += loss.item() * inputs.size(0)
            epoch_total_train_samples += inputs.size(0)

            # Collect true labels and predictions
            _, predicted = torch.max(outputs, dim=1)
            epoch_train_true_labels.extend(targets.cpu().numpy())
            epoch_train_predicted_labels.extend(predicted.cpu().numpy())

        # Calculate training metrics
        avg_epoch_train_loss = round(epoch_total_train_loss / epoch_total_train_samples, 4)
        epoch_train_accuracy = round(accuracy_score(epoch_train_true_labels, epoch_train_predicted_labels), 4)
        epoch_train_f1 = round(f1_score(epoch_train_true_labels, epoch_train_predicted_labels, average='weighted'), 4)
        train_losses.append(avg_epoch_train_loss)
        train_accuracies.append(epoch_train_accuracy)
        train_f1_scores.append(epoch_train_f1)

        # Validation phase
        model.eval()

        epoch_total_val_loss = 0.0
        epoch_total_val_samples = 0
        all_val_true_labels = []
        all_val_predicted_labels = []
```

```python
        with torch.no_grad():
            for inputs, targets in val_loader:
                inputs, targets = inputs.to(device), targets.to(device)
                outputs = model(inputs)

                avg_epoch_val_loss = criterion(outputs, targets)
                epoch_total_val_loss += avg_epoch_val_loss.item() * inputs.size(0)
                epoch_total_val_samples += inputs.size(0)

                # Collect true labels and predictions
                _, predicted = torch.max(outputs, dim=1)
                all_val_true_labels.extend(targets.cpu().numpy())
                all_val_predicted_labels.extend(predicted.cpu().numpy())

        # Calculate validation metrics
        avg_epoch_val_loss = round(epoch_total_val_loss / epoch_total_val_samples, 4)
        epoch_val_accuracy = round(accuracy_score(all_val_true_labels, all_val_predicted_labels), 4)
        epoch_val_f1 = round(f1_score(all_val_true_labels, all_val_predicted_labels, average='weighted'), 4)
        val_losses.append(avg_epoch_val_loss)
        val_accuracies.append(epoch_val_accuracy)
        val_f1_scores.append(epoch_val_f1)

        # Update learning rate
        if scheduler:
            scheduler.step()

        epoch_duration = round(time.time() - start_time)
        print(
            f"Epoch {epoch + 1}/{num_epochs} ({epoch_duration}s) | "
            f"Train: loss {avg_epoch_train_loss}, acc {epoch_train_accuracy*100:.2f}%, f1 {epoch_train_f1*100:.2f}% | "
            f"Val: loss {avg_epoch_val_loss}, acc {epoch_val_accuracy*100:.2f}%, f1 {epoch_val_f1*100:.2f}%"
        )

    # Log undefined predictions
    log_undefined(all_val_predicted_labels, label_map.values())

    # Total training time
    total_training_time = round(time.time() - total_start_time)
    print(f"Total Training Time: {total_training_time}s\n")

    # Save model summary and metrics
    os.makedirs("models", exist_ok=True)
    with open(f"models/{label}.txt", "w", encoding="utf-8") as f:
        f.write(str(summary(model, verbose=0)))

    # Compute and save metrics
    metrics = compute_metrics(all_val_true_labels, all_val_predicted_labels, label_map.values())
    print_metrics(metrics)
    save_metrics(label, metrics)

    # Save model state
    torch.save(model.state_dict(), f"models/{label}.pth")

    # Return training history
    return {
        "num_epochs": num_epochs,
        "train_losses": train_losses,
        "train_accuracies": train_accuracies,
        "train_f1_scores": train_f1_scores,
        "val_losses": val_losses,
        "val_accuracies": val_accuracies,
        "val_f1_scores": val_f1_scores,
    }
```

File: transformer_model.py

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import random
# the self attention is just like described in deep learning by Bishop, so i will not change it.
class SelfAttention(nn.Module):
    def __init__(self, d_model, d_key):
        super().__init__()
        # Three separate linear layers for the queries, keys, and values
```

```
        self.w_q = nn.Linear(d_model, d_key)
        self.w_k = nn.Linear(d_model, d_key)
        self.w_v = nn.Linear(d_model, d_model)
    def forward(self, x):
        q = self.w_q(x)
        k = self.w_k(x)
        v = self.w_v(x)
        # Compute the attention weights
        a = q @ k.transpose(-2, -1) / (k.shape[-1] ** 0.5)
        a = F.softmax(a, dim=-1)
        # Apply the attention weights
        z = a @ v
        return z
# Same as in book, shouldn't need any change
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, d_key, n_heads):
        super().__init__()
        self.heads = nn.ModuleList([SelfAttention(d_model, d_key) for _ in range(n_heads)])
        # Down projection back to model dimension
        self.w_o = nn.Linear(n_heads * d_model, d_model)
    def forward(self, x):
        return self.w_o(torch.cat([h(x) for h in self.heads], dim=-1))
# maybe change siLU activation function?
class TransformerBlock(nn.Module):
    def __init__(self, d_model, d_key, n_heads, dropout1, dropout2, mlp_factor=4, ):
        super().__init__()
        # We need to init two layer norms because they have parameters
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = MultiHeadSelfAttention(d_model, d_key, n_heads)
        self.ln2 = nn.LayerNorm(d_model)
        # a feedforward module
        if dropout1 > 0:
            self.mlp = nn.Sequential(
                nn.Linear(d_model, mlp_factor * d_model),
                nn.Dropout(p = dropout1),
                nn.SiLU(),  # Swish activation function, f(x) = x * sigmoid(x)
                nn.Linear(mlp_factor * d_model, d_model),
                nn.Dropout(p = dropout2)
            )
        else:
            self.mlp = nn.Sequential(
                nn.Linear(d_model, mlp_factor * d_model),
                nn.SiLU(),  # Swish activation function, f(x) = x * sigmoid(x)
                nn.Linear(mlp_factor * d_model, d_model),
                nn.SiLU(),
                nn.Linear(d_model, mlp_factor * d_model),
                nn.SiLU(),
                nn.Linear(d_model, mlp_factor * d_model),
            )
    def forward(self, x):
        # Residual connections and pre-layernorm
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x
class TransformerClassifier(nn.Module):
    def __init__(self, n_embeds, n_classes, d_model=256, d_key=64, n_heads=2, mlp_factor=4, n_layers=2, device = "cpu",
        super().__init__()
        self.device = device
        self.d_model = d_model
        self.token_embedding = nn.Embedding(n_embeds, d_model)
        self.transformer_model = nn.Sequential(*[TransformerBlock(d_model, d_key, n_heads, dropout1, dropout2, mlp_fact
        self.final_layer_norm = nn.LayerNorm(d_model)
        self.classifier = nn.Sequential(nn.Linear(d_model, d_model), nn.SiLU(), nn.Linear(d_model, n_classes))

    def sinusoidalPositionEncoding(self, input):
        # create empty matrix
        r_n_matrix = torch.empty((input.size(dim=1), self.d_model))
        r_n_matrix = r_n_matrix.to(self.device)
        # fill all areas of empty matrix
        for n in range(input.size(dim=1)):
            for i in range(self.d_model):
                if i % 2 == 0:
                    r_n_matrix[n, i] =  math.sin(n / 10000 ** (i / self.d_model))
                if i % 2 == 1:
                    r_n_matrix[n, i] = math.cos(n / 10000 ** (i / self.d_model))
        # add with input
        input_hat = input + r_n_matrix
        # return modified input
```

```
        return input_hat

    def forward(self, x):
        e = self.token_embedding(x)
        # sinusoidal positional encoding
        s = self.sinusoidalPositionEncoding(e)
        h = self.transformer_model(s)
        h = h.mean(dim=1) # Average pooling on the sequence dimension
        y = self.classifier(self.final_layer_norm(h))
        return y
```

B Notebook

notebook

November 29, 2024

0.1 Libraries

[4]:
```
%load_ext autoreload
%autoreload 2
```

[5]:
```
from collections import Counter
from collections import defaultdict
import itertools
import json
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import nltk
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
nltk.download('wordnet')  # downloads WordNet data
nltk.download('omw-1.4')  # downloads additional wordnet data for lemmatization
nltk.download('stopwords')  # downloads stopwords (if not already downloaded)
from sklearn.metrics import (
    accuracy_score,
    f1_score,
)
from wordcloud import WordCloud
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from datasets import Dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    Trainer,
    TrainingArguments,
)

from emotion_dataset import EmotionDataset
from loader import loader
from train_model import train_model
from plot_scores import plot_scores
from predict import predict
from predict_on_fly import predict_on_fly
from metrics import compute_metrics, print_metrics, save_metrics
import transformer_model
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\difj6\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\difj6\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\difj6\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
c:\Users\difj6\OneDrive - Syddansk Universitet\Documents\Uni\7. semester\DM873
Deep learning\Project2\.venv\lib\site-packages\tqdm\auto.py:21: TqdmWarning:
IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

0.2 Device

[6]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using {device}")
```

```
Using cuda
```

0.3 Load and save dataset

[7]:
```
train_csv_path = "data/train.csv"
val_csv_path = "data/val.csv"
test_csv_path = "data/test.csv"

train_df, val_df, test_df = loader(train_csv_path, val_csv_path, test_csv_path)
```

1 Task 1: Data Preparation

1.1 Data set

[8]:
```
print(f"# train sentences: {len(train_df)}")
print(f"# validation sentences: {len(val_df)}")
print(f"# test sentences: {len(test_df)}")

print(train_df)
```

```
# train sentences: 16000
# validation sentences: 2000
# test sentences: 2000
                                                    text  label label_name
0                                i didnt feel humiliated      0    sadness
1      i can go from feeling so hopeless to so damned...      0    sadness
2       im grabbing a minute to post i feel greedy wrong      3      anger
3      i am ever feeling nostalgic about the fireplac...      2       love
4                                   i am feeling grouchy      3      anger
...                                                  ...    ...        ...
15995  i just had a very brief time in the beanbag an...      0    sadness
15996  i am now turning and i feel pathetic that i am...      0    sadness
15997                     i feel strong and good overall      1        joy
15998  i feel like this was such a rude comment and i...      3      anger
15999  i know a lot but i feel so stupid because i ca...      0    sadness

[16000 rows x 3 columns]
```

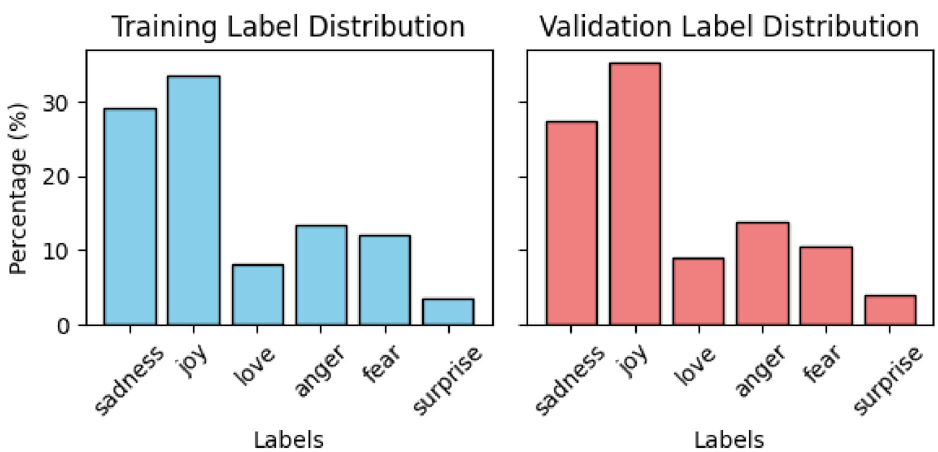
1.2 Step 1: Dataset Preparation

1.2.1 Label distribution

[9]:
```
# Calculate label distributions and percentages for training and validation sets
def calculate_distribution(df):
    label_distribution = df["label_name"].value_counts().reset_index()
    label_distribution.columns = ["label_name", "count"]
    label_distribution["percentage"] = round((label_distribution["count"] / label_distribution["count"].
↪sum()) * 100, 2)
    return label_distribution

# Calculate distributions
train_distribution = calculate_distribution(train_df)
val_distribution = calculate_distribution(val_df)

# Merge with label mapping for alignment
label_map_df = train_df[["label", "label_name"]].drop_duplicates().sort_values("label")
label_map = dict(zip(label_map_df["label"], label_map_df["label_name"]))
train_labels_df = label_map_df.merge(train_distribution, on="label_name", how="left")
val_labels_df = label_map_df.merge(val_distribution, on="label_name", how="left")

# Print training label mapping and distribution
num_classes = len(label_map_df)
print(f"Number of classes: {num_classes}")
```

```
print("Training Label Mapping and Distribution:")
print(train_labels_df.to_string(index=False))

# Plot side-by-side
fig, axes = plt.subplots(1, 2, figsize=(6, 3), sharey=True)

# Training set
axes[0].bar(train_labels_df["label_name"], train_labels_df["percentage"], color="skyblue",␣
↪edgecolor="black")
axes[0].set_title("Training Label Distribution")
axes[0].set_xlabel("Labels")
axes[0].set_ylabel("Percentage (%)")
axes[0].tick_params(axis='x', rotation=45)

# Validation set
axes[1].bar(val_labels_df["label_name"], val_labels_df["percentage"], color="lightcoral", edgecolor="black")
axes[1].set_title("Validation Label Distribution")
axes[1].set_xlabel("Labels")
axes[1].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()
```

```
Number of classes: 6
Training Label Mapping and Distribution:
 label label_name  count  percentage
     0    sadness   4666       29.16
     1        joy   5362       33.51
     2       love   1304        8.15
     3      anger   2159       13.49
     4       fear   1937       12.11
     5   surprise    572        3.58
```



1.3 Step 2: Tokenizing

1.3.1 Tokenizer

[10]:
```
tokenizer = RegexpTokenizer(r"[a-zA-Z]+|[!?'´`]+") # sequence that don't match the pattern act as␣
↪separators.
example_sentence = "This?.is,a:cu123stom;tokenization example!<"
example_tokens = tokenizer.tokenize(example_sentence)
print(example_tokens)
```

```
['This', '?', 'is', 'a', 'cu', 'stom', 'tokenization', 'example', '!']
```

1.3.2 Tokenize each split

[11]:
```
train_df["tokens"] = train_df["text"].str.lower().apply(tokenizer.tokenize)
val_df["tokens"] = val_df["text"].str.lower().apply(tokenizer.tokenize)
test_df["tokens"] = test_df["text"].str.lower().apply(tokenizer.tokenize)

train_vocab = set(token for tokens in train_df["tokens"] for token in tokens)

print(f"# words in train vocab: {len(train_vocab)}")
```

```
# words in train vocab: 15212
```

1.3.3 Word frequency

[12]:
```
def get_top_words_per_class(tokens_in, top_n=10):
    tokens_by_class = defaultdict(list)
    for tokens, label in zip(tokens_in, train_df["label_name"]):
        tokens_by_class[label].append(tokens)
    tokens_by_class = dict(tokens_by_class)
    results = []
    for label_name, tokens in tokens_by_class.items():
        flat_tokens = list(itertools.chain.from_iterable(tokens))
        most_common = Counter(flat_tokens).most_common(top_n)
        for word, count in most_common:
            results.append({"Label_name": label_name, "Word": word, "Count": count})
    return pd.DataFrame(results)

print(get_top_words_per_class(train_df["tokens"], top_n=10).to_string(index=False))
```

```
Label_name    Word  Count
   sadness       i   7635
   sadness    feel   3299
   sadness     and   2692
   sadness      to   2335
   sadness     the   2155
   sadness       a   1656
   sadness feeling   1523
   sadness      of   1422
   sadness    that   1299
   sadness      my   1245
     anger       i   3576
     anger    feel   1459
     anger     and   1258
     anger      to   1162
     anger     the   1109
     anger       a    791
     anger feeling    721
     anger    that    705
     anger      of    630
     anger      my    573
      love       i   2120
      love    feel    929
      love     and    902
      love      to    860
      love     the    780
      love       a    571
      love      of    482
      love    that    460
      love      my    399
      love feeling    378
  surprise       i    927
  surprise    feel    356
  surprise     and    354
  surprise     the    335
  surprise      to    267
  surprise       a    256
```

```
surprise    that   212
surprise feeling   209
surprise      of   191
surprise      my   163
    fear       i  3083
    fear    feel  1212
    fear      to  1116
    fear     and  1110
    fear     the  1000
    fear       a   806
    fear feeling   742
    fear      of   614
    fear    that   531
    fear      my   525
     joy       i  8518
     joy    feel  3928
     joy     and  3273
     joy      to  3232
     joy     the  2991
     joy       a  2120
     joy    that  1905
     joy      of  1651
     joy feeling  1539
     joy      my  1378
```

1.3.4 Data cleaning (remove stopwords and stem)

[13]:
```
stemmer = PorterStemmer()
stop_words = set(stopwords.words("english"))

def preprocess_tokens(tokens):
    return [stemmer.stem(word) for word in tokens if word not in stop_words]

train_df["processed_tokens"] = train_df["tokens"].apply(preprocess_tokens)
val_df["processed_tokens"] = val_df["tokens"].apply(preprocess_tokens)
test_df["processed_tokens"] = test_df["tokens"].apply(preprocess_tokens)

print(get_top_words_per_class(train_df["processed_tokens"], top_n=10).to_string(index=False))
```

```
Label_name      Word  Count
   sadness      feel   4994
   sadness      like    881
   sadness        im    683
   sadness      know    297
   sadness       get    284
   sadness    realli    276
   sadness      time    271
   sadness      make    245
   sadness      want    244
   sadness        go    235
     anger      feel   2261
     anger      like    391
     anger        im    342
     anger       get    175
     anger      time    140
     anger      want    133
     anger     irrit    128
     anger    realli    124
     anger      know    122
     anger      hate    113
      love      feel   1406
      love      like    366
      love      love    277
      love        im    193
      love   support    103
      love    realli     92
      love      know     89
      love      want     89
```

```
    love      time    82
    love      care    82
surprise      feel   601
surprise      amaz   107
surprise      like    92
surprise        im    91
surprise   impress    63
surprise overwhelm    58
surprise     weird    57
surprise   surpris    56
surprise    curiou    54
surprise     funni    49
    fear      feel  2025
    fear        im   322
    fear      like   264
    fear     littl   149
    fear        go   139
    fear      know   136
    fear       bit   118
    fear      want   113
    fear      time   110
    fear       get   107
     joy      feel  5674
     joy      like  1023
     joy        im   799
     joy      make   381
     joy      time   334
     joy       get   322
     joy        go   315
     joy    realli   309
     joy      want   272
     joy      know   261
```

1.3.5 Frequency of words

[14]:
```python
word_freq = Counter(itertools.chain.from_iterable(train_df["processed_tokens"]))
pd.DataFrame(word_freq.items(), columns=["Word", "Count"]).sort_values(by="Count", ascending=False).
↪to_csv("results/word_frequencies.csv", index=False)
print(pd.DataFrame(word_freq.most_common(100), columns=["Word", "Count"]).to_string(index=False))
```

```
  Word  Count
  feel  16961
  like   3017
    im   2430
   get    981
  time    979
realli    942
  know    938
  make    935
    go    882
  want    867
  love    805
 littl    736
 think    736
   day    675
 thing    672
 peopl    664
   one    647
 would    646
  even    600
 still    598
   ive    587
  life    555
   way    528
  need    521
   bit    521
someth    514
  much    496
```

dont	482
work	471
could	453
say	450
start	445
look	423
see	419
back	414
tri	410
good	408
pretti	392
right	357
alway	356
come	351
help	342
friend	340
also	337
year	336
today	332
use	326
take	317
around	315
person	303
cant	301
made	296
hate	285
well	279
though	274
happi	274
didnt	272
got	271
write	270
live	268
felt	266
lot	264
never	264
thought	263
hope	261
someon	259
find	259
everi	254
quit	250
read	246
less	246
sure	240
enough	238
week	236
give	234
mani	232
kind	230
home	227
away	226
support	224
long	222
ever	221
anyth	220
actual	220
talk	215
better	213
keep	212
left	211
let	210
everyth	210
without	209
rememb	209
last	207
care	205
tell	205

```
  world    205
 wonder    204
sometim    201
    new    199
   http    199
```

1.3.6 Remove additional words

[15]:
```python
additional_words_to_remove = ["feel", "realli", "im", "know", "also", "http"]

def remove_additional_words(tokens):
    return [word for word in tokens if word not in additional_words_to_remove]

train_df["processed_tokens"] = train_df["processed_tokens"].apply(remove_additional_words)
val_df["processed_tokens"] = val_df["processed_tokens"].apply(remove_additional_words)
test_df["processed_tokens"] = test_df["processed_tokens"].apply(remove_additional_words)

train_df["processed_text"] = train_df["processed_tokens"].apply(" ".join)
val_df["processed_text"] = val_df["processed_tokens"].apply(" ".join)
test_df["processed_text"] = test_df["processed_tokens"].apply(" ".join)

print(get_top_words_per_class(train_df["processed_tokens"], top_n=10).to_string(index=False))
```

```
Label_name      Word  Count
   sadness      like    881
   sadness       get    284
   sadness      time    271
   sadness      make    245
   sadness      want    244
   sadness        go    235
   sadness       day    224
   sadness     thing    221
   sadness       ive    217
   sadness     think    212
     anger      like    391
     anger       get    175
     anger      time    140
     anger      want    133
     anger     irrit    128
     anger      hate    113
     anger     thing    109
     anger      make    108
     anger        go    108
     anger     think    105
      love      like    366
      love      love    277
      love   support    103
      love      want     89
      love      time     82
      love      care     82
      love      long     72
      love       one     70
      love       get     70
      love     sweet     69
  surprise      amaz    107
  surprise      like     92
  surprise   impress     63
  surprise overwhelm     58
  surprise     weird     57
  surprise   surpris     56
  surprise    curiou     54
  surprise     funni     49
  surprise    strang     46
  surprise     shock     46
      fear      like    264
      fear     littl    149
      fear        go    139
      fear       bit    118
```

```
fear      want    113
fear      time    110
fear       get    107
fear      make    105
fear     think     94
fear     peopl     90
 joy      like   1023
 joy      make    381
 joy      time    334
 joy       get    322
 joy        go    315
 joy      want    272
 joy      love    257
 joy       day    241
 joy     think    233
 joy       one    211
```

1.3.7 WordCloud

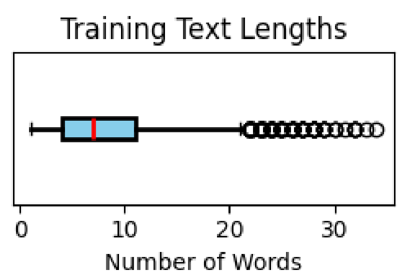
[16]:
```python
wordcloud = WordCloud(width=400, height=200, background_color="white").generate(" ".join(list(itertools.
↪chain.from_iterable(train_df["processed_tokens"]))))
plt.figure(figsize=(5, 3))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

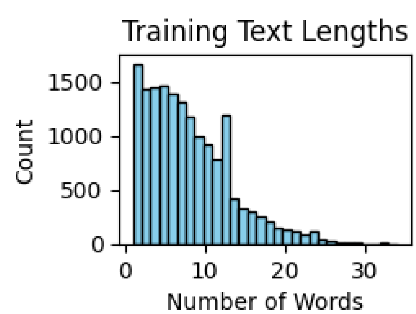
1.3.8 Sentence length distribution

[17]:
```python
train_lengths = [len(tokens) for tokens in train_df["processed_tokens"]]
mean_length = np.mean(train_lengths)
std_dev = np.std(train_lengths)

print(f"Length range for train: from {min(train_lengths)} to {max(train_lengths)} words")
print(f"Mean length for train: {mean_length:.0f} words")
print(f"Standard deviation for train: {std_dev:.0f}")

# Plot boxplot
plt.figure(figsize=(3, 1.2))
plt.boxplot(
    train_lengths,
    vert=False,
    patch_artist=True,
    boxprops=dict(facecolor="skyblue", linewidth=2), # larger, colored box
    whiskerprops=dict(linewidth=2),  # Thicker whiskers
    medianprops=dict(color="red", linewidth=2), # highlight the median
)
plt.title("Training Text Lengths")
```

```
plt.xlabel("Number of Words")
plt.yticks([])
plt.show()

# Plot distribution of lengths
plt.figure(figsize=(2.3, 1.5))
plt.hist(train_lengths, bins=30, color="skyblue", edgecolor="black")
plt.title("Training Text Lengths")
plt.xlabel("Number of Words")
plt.ylabel("Count")
plt.show()
```

```
Length range for train: from 1 to 34 words
Mean length for train: 8 words
Standard deviation for train: 5
```





1.3.9 Set max and min length

[18]:
```
# Set max length for the model. If sentence is longer, truncate it. If shorter, pad it.
# Set min length to remove very short sentences from the training set.
max_length = 10
min_length = 3

print(f"# train sentences before filtering: {len(train_df)}")
train_df = train_df[train_df["processed_tokens"].apply(len) >= min_length]
print(f"# train sentences after filtering: {len(train_df)}")
```

```
# train sentences before filtering: 16000
# train sentences after filtering: 14331
```

1.4 Step 3: Build a vocabulary

[19]:
```
vocab = {"<PAD>": 0, "<UNK>": 1}
for tokens in train_df["processed_tokens"]:
    for token in tokens:
        if token not in vocab:
            vocab[token] = len(vocab)

vocab_size = len(vocab)
print(f"Vocabulary size: {vocab_size}")

reverse_vocab = {v: k for k, v in vocab.items()}
```

```
Vocabulary size: 10336
```

1.5 Step 4: Encode all texts with the vocabulary

[20]:
```
def encode(tokens): # i.e. words to integers
    return [vocab[token] if token in vocab else 1 for token in tokens]

train_df["encoded"] = train_df["processed_tokens"].apply(encode)
val_df["encoded"] = val_df["processed_tokens"].apply(encode)
test_df["encoded"] = test_df["processed_tokens"].apply(encode)
```

1.6 Step 5: Maximum sequence length

[21]:
```
def pad(sequence):
    return sequence[:max_length] + [0] * (max_length - len(sequence))

train_df["padded"] = train_df["encoded"].apply(pad)
val_df["padded"] = val_df["encoded"].apply(pad)
test_df["padded"] = test_df["encoded"].apply(pad)
```

2 Task 2: RNN model

2.1 Model class

[72]:
```
class RNN_model(nn.Module):
    def __init__(self, type, vocab_size, embedding_dim, hidden_size, num_classes, padding_idx=0,␣
↪num_layers=1, dropout_rnn=0, dropout_fc=0):
        super(RNN_model, self).__init__()
        # embedding_dim: size of each embedding vector
        # hidden_size: number of features in the hidden state
        # num_layers: number of recurrent layers
        # bias: introduces a bias
        # batch_first: input and output tensors are provided as (batch, seq, feature)
        # dropout: if non-zero, introduces a dropout layer on the outputs of each RNN layer except the last␣
↪layer

        self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim,␣
↪padding_idx=padding_idx)
        if type == "RNN":
            self.rnn = nn.RNN(input_size=embedding_dim, hidden_size=hidden_size, num_layers=num_layers,␣
↪bias=True, batch_first=True, dropout=dropout_rnn, nonlinearity="tanh")
        elif type == "GRU":
            self.rnn = nn.GRU(input_size=embedding_dim, hidden_size=hidden_size, num_layers=num_layers,␣
↪bias=True, batch_first=True, dropout=dropout_rnn)
        elif type == "LSTM":
            self.rnn = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_size, num_layers=num_layers,␣
↪bias=True, batch_first=True, dropout=dropout_rnn)
        self.fc = nn.Linear(in_features=hidden_size, out_features=num_classes)
        self.dropout = nn.Dropout(p=dropout_fc)

    def forward(self, x):
        x = self.embedding(x)
        x, _ = self.rnn(x)
```

```
        x = x[:, -1, :]  # extract last hidden state for each sequence
        x = self.dropout(x) # apply dropout to the last hidden state
        x = self.fc(x) # pass last hidden state through the fully connected layer
        return x
```

2.2 Hyper parameter tuning

[73]:
```
train_dataset = EmotionDataset(train_df["padded"].tolist(), train_df["label"].tolist())
val_dataset = EmotionDataset(val_df["padded"].tolist(), val_df["label"].tolist())
test_dataset = EmotionDataset(test_df["padded"].tolist(), test_df["label"].tolist())
```

[ ]:
```
def grid_search(vocab_size, num_classes, train_dataset, val_dataset, label_map, device):
    param_grid = {
        # The best hyperparameters found is commented out
        "type": ["LSTM", "GRU", "RNN"], # "LSTM"
        "embedding_dim": [100, 75], # 75
        "hidden_size": [512, 256], # 256
        "layers": [1, 2], # 1
        "dropout_rnn": [0.0, 0.2], # 0.0
        "dropout_fc": [0.4, 0.6], # 0.4
        "learning_rate": [0.001, 0.0005], # 0.0001
        "reg_lambda": [0.0001, 0.00005], # 0.0001
        "batch_size": [16, 32], # 16
    }

    # Generate all combinations of hyperparameters
    keys, values = zip(*param_grid.items())
    configs = [dict(zip(keys, v)) for v in itertools.product(*values)]
    total_configs = len(configs)

    best_val_f1_score = 0
    results_list = []
    for i, config in enumerate(configs):
        label = f"model_{i}"
        print(f"Training model: {label} ({i+1}/{total_configs})")
        print(json.dumps(config, indent=4))

        train_loader = DataLoader(train_dataset, batch_size=config["batch_size"], shuffle=True)
        val_loader = DataLoader(val_dataset, batch_size=config["batch_size"], shuffle=False)

        # Model
        model = RNN_model(
            type=config["type"],
            vocab_size=vocab_size,
            embedding_dim=config["embedding_dim"],
            hidden_size=config["hidden_size"],
            num_classes=num_classes,
            num_layers=config["layers"],
            dropout_rnn=config["dropout_rnn"],
            dropout_fc=config["dropout_fc"],
        )

        # Train
        results = train_model(
            label=label,
            model=model,
            train_loader=train_loader,
            val_loader=val_loader,
            label_map=label_map,
            device=device,
            optimizer_type="AdamW",
            learning_rate=config["learning_rate"],
            reg_type="L2",
            reg_lambda=config["reg_lambda"],
            num_epochs=10
        )
```

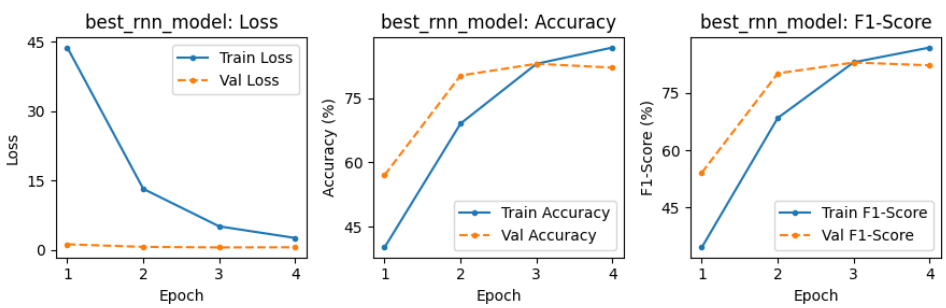
```
        # Track the best model
        current_val_f1_score = max(results["val_f1_scores"])
        if current_val_f1_score > best_val_f1_score:
            best_val_f1_score = current_val_f1_score
            best_model_label = label
            print(f"New best model found: {best_model_label} with val f1 score: {best_val_f1_score:.4f}")

        # Add results to list
        results["config"] = config
        results_list.append(results)

    # Save results
    with open("results/grid_search_results.json", "w") as f:
        json.dump(results_list, f, indent=4)

    print(f"Best model found: {best_model_label}")

grid_search(vocab_size, num_classes, train_dataset, val_dataset, label_map, device)
```

2.3 Best RNN model

[75]:
```
label = "best_rnn_model"
best_batch_size = 16

train_loader = DataLoader(train_dataset, batch_size=best_batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=best_batch_size, shuffle=False)

best_rnn_model = RNN_model(
    type="LSTM",
    vocab_size=vocab_size,
    embedding_dim=75,
    hidden_size=256,
    num_classes=num_classes,
    num_layers=1,
    dropout_rnn=0.0,
    dropout_fc=0.4,
)

best_rnn_results = train_model(
    label=label,
    model=best_rnn_model,
    train_loader=train_loader,
    val_loader=val_loader,
    label_map=label_map,
    device=device,
    optimizer_type="AdamW",
    learning_rate=0.001,
    reg_type="L2",
    reg_lambda=0.0001,
    num_epochs=4
)

plot_scores(best_rnn_results, label)
```

```
Epoch 1/4 (10s) | Train: loss 43.7021, acc 40.13%, f1 34.61% | Val: loss 1.1907,
acc 57.05%, f1 54.13%
Epoch 2/4 (7s) | Train: loss 13.1735, acc 69.10%, f1 68.43% | Val: loss 0.6395,
acc 80.35%, f1 80.08%
Epoch 3/4 (7s) | Train: loss 5.0673, acc 83.09%, f1 82.97% | Val: loss 0.5178,
acc 83.10%, f1 82.92%
Epoch 4/4 (7s) | Train: loss 2.5631, acc 86.90%, f1 86.83% | Val: loss 0.551,
acc 82.20%, f1 82.18%
Total Training Time: 32s

accuracy: 0.822
f1: 0.8218
precision: 0.8257
```

```
recall: 0.822
classification report:
              precision    recall  f1-score   support

     sadness       0.80      0.91      0.85       550
         joy       0.89      0.83      0.86       704
        love       0.69      0.69      0.69       178
       anger       0.86      0.76      0.80       275
        fear       0.79      0.80      0.79       212
    surprise       0.75      0.77      0.76        81

    accuracy                           0.82      2000
   macro avg       0.79      0.79      0.79      2000
weighted avg       0.83      0.82      0.82      2000

confusion matrix:
[502, 15, 7, 8, 16, 2]
[54, 581, 42, 10, 10, 7]
[15, 29, 122, 6, 3, 3]
[41, 14, 3, 208, 8, 1]
[16, 6, 4, 9, 169, 8]
[3, 5, 0, 2, 9, 62]
```



3 Task 3: Transformer model

[76]:
```
batch_size = 32
train_dataset = EmotionDataset(train_df["padded"].tolist(), train_df["label"].tolist())
val_dataset = EmotionDataset(val_df["padded"].tolist(), val_df["label"].tolist())
test_dataset = EmotionDataset(test_df["padded"].tolist(), test_df["label"].tolist())
train_labels = train_df["label"].tolist()
val_labels = val_df["label"].tolist()
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
n_embeds = 10336
num_epochs = 50
```

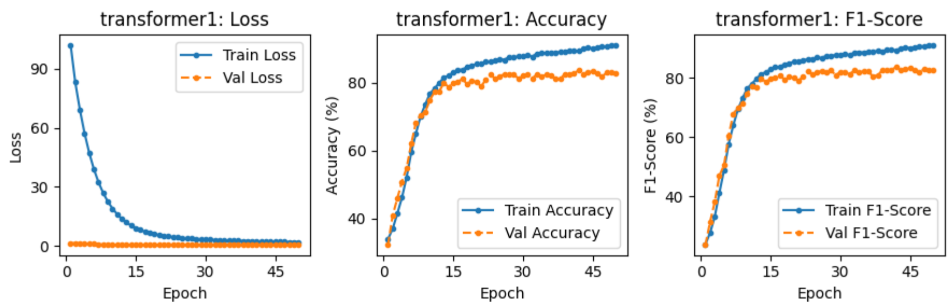
[81]:
```
config = {
        "label": "transformer1",
        "d_key": 64,
        "n_heads": 8,
        "mlp_factor": 4,
        "d_model": 128,
        "n_layers": 6,
        "dropout1": 0.5,
        "dropout2": 0.25,
        "optimizer_type": "Adam",
        "learning_rate": 0.0003,
        "weight_decay": 0,
```

```
        "reg_type": "L1",
        "reg_lambda": 1e-4,
    }
print(f"Training model: {config['label']}")
model = transformer_model.TransformerClassifier(n_embeds=n_embeds, n_classes=6, d_model=config["d_model"],␣
↪d_key=config["d_key"], n_heads=config["n_heads"], mlp_factor=config["mlp_factor"],␣
↪n_layers=config["n_layers"], device = device, dropout1=config["dropout1"], dropout2=config["dropout2"])
results = train_model(
    label=config["label"],
    model=model,
    train_loader=train_loader,
    val_loader=val_loader,
    label_map=label_map,
    device=device,
    optimizer_type=config["optimizer_type"],
    learning_rate=config["learning_rate"],
    weight_decay=config["weight_decay"],
    reg_type=config["reg_type"],
    reg_lambda=config["reg_lambda"],
    num_epochs=num_epochs,
)
plot_scores(results, config["label"])
```

```
Training model: transformer1
Epoch 1/50 (89s) | Train: loss 102.0006, acc 33.86%, f1 23.71% | Val: loss
1.5793, acc 32.10%, f1 23.49%
Epoch 2/50 (76s) | Train: loss 82.9869, acc 36.89%, f1 27.59% | Val: loss
1.5281, acc 40.85%, f1 31.31%
Epoch 3/50 (76s) | Train: loss 69.0236, acc 41.38%, f1 33.03% | Val: loss
1.4376, acc 45.90%, f1 38.03%
Epoch 4/50 (80s) | Train: loss 57.1575, acc 46.12%, f1 41.04% | Val: loss
1.3466, acc 50.80%, f1 46.88%
Epoch 5/50 (77s) | Train: loss 47.2545, acc 52.08%, f1 48.79% | Val: loss
1.1978, acc 54.85%, f1 50.44%
Epoch 6/50 (80s) | Train: loss 39.011, acc 59.49%, f1 57.57% | Val: loss 1.062,
acc 62.00%, f1 60.50%
Epoch 7/50 (79s) | Train: loss 32.3509, acc 65.00%, f1 63.95% | Val: loss
0.9193, acc 68.25%, f1 67.81%
Epoch 8/50 (77s) | Train: loss 26.9274, acc 70.02%, f1 69.34% | Val: loss
0.8478, acc 70.40%, f1 69.82%
Epoch 9/50 (76s) | Train: loss 22.5769, acc 73.58%, f1 73.13% | Val: loss
0.8064, acc 71.35%, f1 71.28%
Epoch 10/50 (76s) | Train: loss 19.0561, acc 76.75%, f1 76.42% | Val: loss
0.712, acc 74.90%, f1 74.65%
Epoch 11/50 (76s) | Train: loss 16.2046, acc 78.24%, f1 77.94% | Val: loss
0.6758, acc 77.20%, f1 77.05%
Epoch 12/50 (76s) | Train: loss 13.8662, acc 79.91%, f1 79.69% | Val: loss
0.6526, acc 77.35%, f1 76.76%
Epoch 13/50 (76s) | Train: loss 11.9736, acc 81.45%, f1 81.25% | Val: loss
0.6205, acc 79.95%, f1 79.66%
Epoch 14/50 (77s) | Train: loss 10.4343, acc 82.03%, f1 81.87% | Val: loss
0.6295, acc 78.65%, f1 78.57%
Epoch 15/50 (77s) | Train: loss 9.1926, acc 82.99%, f1 82.84% | Val: loss
0.5959, acc 79.95%, f1 79.67%
Epoch 16/50 (76s) | Train: loss 8.1612, acc 83.69%, f1 83.55% | Val: loss
0.5867, acc 80.30%, f1 80.02%
Epoch 17/50 (77s) | Train: loss 7.3331, acc 83.83%, f1 83.73% | Val: loss
0.5698, acc 81.00%, f1 80.82%
Epoch 18/50 (76s) | Train: loss 6.6518, acc 84.61%, f1 84.49% | Val: loss
0.6136, acc 79.55%, f1 79.36%
Epoch 19/50 (76s) | Train: loss 6.104, acc 84.82%, f1 84.73% | Val: loss 0.5767,
acc 80.55%, f1 80.34%
Epoch 20/50 (76s) | Train: loss 5.6224, acc 85.53%, f1 85.45% | Val: loss
0.5924, acc 80.15%, f1 79.88%
Epoch 21/50 (76s) | Train: loss 5.2473, acc 85.56%, f1 85.48% | Val: loss
0.6482, acc 78.85%, f1 78.85%
Epoch 22/50 (76s) | Train: loss 4.9044, acc 86.11%, f1 86.02% | Val: loss
0.5677, acc 80.80%, f1 80.52%
```

```
Epoch 23/50 (76s) | Train: loss 4.6268, acc 86.14%, f1 86.07% | Val: loss
0.5594, acc 82.25%, f1 82.19%
Epoch 24/50 (76s) | Train: loss 4.3677, acc 86.46%, f1 86.37% | Val: loss
0.5706, acc 81.20%, f1 80.95%
Epoch 25/50 (78s) | Train: loss 4.1486, acc 86.97%, f1 86.90% | Val: loss
0.5459, acc 81.90%, f1 81.75%
Epoch 26/50 (76s) | Train: loss 3.9806, acc 86.69%, f1 86.62% | Val: loss 0.542,
acc 82.40%, f1 82.38%
Epoch 27/50 (77s) | Train: loss 3.7892, acc 87.52%, f1 87.46% | Val: loss 0.542,
acc 82.30%, f1 81.97%
Epoch 28/50 (76s) | Train: loss 3.6444, acc 87.35%, f1 87.29% | Val: loss
0.5416, acc 82.55%, f1 82.39%
Epoch 29/50 (76s) | Train: loss 3.4955, acc 87.80%, f1 87.74% | Val: loss
0.5703, acc 81.15%, f1 80.88%
Epoch 30/50 (77s) | Train: loss 3.389, acc 87.71%, f1 87.65% | Val: loss 0.546,
acc 82.00%, f1 81.85%
Epoch 31/50 (76s) | Train: loss 3.2559, acc 88.16%, f1 88.10% | Val: loss
0.5646, acc 82.30%, f1 81.96%
Epoch 32/50 (76s) | Train: loss 3.1771, acc 87.64%, f1 87.59% | Val: loss
0.5844, acc 81.15%, f1 81.21%
Epoch 33/50 (76s) | Train: loss 3.0501, acc 88.59%, f1 88.54% | Val: loss
0.5427, acc 82.45%, f1 82.40%
Epoch 34/50 (76s) | Train: loss 2.9593, acc 88.60%, f1 88.55% | Val: loss
0.5736, acc 81.90%, f1 81.82%
Epoch 35/50 (76s) | Train: loss 2.8801, acc 88.61%, f1 88.56% | Val: loss
0.5598, acc 82.25%, f1 82.16%
Epoch 36/50 (77s) | Train: loss 2.8071, acc 88.83%, f1 88.78% | Val: loss
0.5595, acc 82.15%, f1 82.04%
Epoch 37/50 (77s) | Train: loss 2.7384, acc 88.84%, f1 88.79% | Val: loss
0.6097, acc 81.05%, f1 80.46%
Epoch 38/50 (76s) | Train: loss 2.659, acc 89.01%, f1 88.96% | Val: loss 0.5831,
acc 81.10%, f1 80.82%
Epoch 39/50 (77s) | Train: loss 2.5879, acc 88.94%, f1 88.89% | Val: loss
0.5631, acc 82.45%, f1 82.42%
Epoch 40/50 (77s) | Train: loss 2.5379, acc 89.44%, f1 89.40% | Val: loss 0.589,
acc 82.45%, f1 82.54%
Epoch 41/50 (77s) | Train: loss 2.4701, acc 89.35%, f1 89.31% | Val: loss
0.5726, acc 82.80%, f1 82.71%
Epoch 42/50 (76s) | Train: loss 2.4347, acc 89.32%, f1 89.28% | Val: loss
0.5533, acc 83.55%, f1 83.54%
Epoch 43/50 (77s) | Train: loss 2.3524, acc 89.97%, f1 89.93% | Val: loss
0.5814, acc 82.50%, f1 82.40%
Epoch 44/50 (76s) | Train: loss 2.3024, acc 90.38%, f1 90.34% | Val: loss
0.5854, acc 83.15%, f1 83.07%
Epoch 45/50 (77s) | Train: loss 2.2664, acc 89.99%, f1 89.95% | Val: loss
0.5644, acc 83.30%, f1 83.14%
Epoch 46/50 (76s) | Train: loss 2.2204, acc 90.38%, f1 90.34% | Val: loss
0.5628, acc 82.60%, f1 82.43%
Epoch 47/50 (76s) | Train: loss 2.167, acc 90.64%, f1 90.61% | Val: loss 0.606,
acc 81.65%, f1 81.72%
Epoch 48/50 (76s) | Train: loss 2.1392, acc 90.64%, f1 90.62% | Val: loss
0.5831, acc 83.00%, f1 82.94%
Epoch 49/50 (76s) | Train: loss 2.0786, acc 91.10%, f1 91.07% | Val: loss
0.5902, acc 82.90%, f1 82.65%
Epoch 50/50 (76s) | Train: loss 2.056, acc 90.95%, f1 90.92% | Val: loss 0.594,
acc 82.80%, f1 82.62%
Total Training Time: 3843s

accuracy: 0.828
f1: 0.8262
precision: 0.8267
recall: 0.828
classification report:
              precision    recall  f1-score   support

     sadness       0.88      0.85      0.87       550
         joy       0.83      0.89      0.86       704
        love       0.74      0.60      0.66       178
```

```
       anger       0.81      0.82      0.81       275
        fear       0.79      0.81      0.80       212
    surprise       0.74      0.68      0.71        81

    accuracy                           0.83      2000
   macro avg       0.80      0.78      0.79      2000
weighted avg       0.83      0.83      0.83      2000

confusion matrix:
[470, 34, 7, 18, 18, 3]
[23, 628, 24, 17, 5, 7]
[8, 51, 107, 8, 3, 1]
[16, 17, 4, 225, 12, 1]
[14, 12, 2, 6, 171, 7]
[2, 12, 0, 4, 8, 55]
```



Train multiple:

```
[ ]: configurations = [
         #heads 16
         {
             "label": "transformer1",
             "d_key": 64,
             "n_heads": 16,
             "mlp_factor": 4,
             "d_model": 128,
             "n_layers": 6,
             "dropout1": 0.5,
             "dropout2": 0.25,
             "optimizer_type": "Adam",
             "learning_rate": 0.001,
             "weight_decay": 0,
             "reg_type": "L2",
             "reg_lambda": 1e-4,
         },
         # mlp_factor 2
         {
             "label": "transformer1",
             "d_key": 64,
             "n_heads": 8,
             "mlp_factor": 2,
             "d_model": 128,
             "n_layers": 6,
             "dropout1": 0.5,
             "dropout2": 0.25,
             "optimizer_type": "Adam",
             "learning_rate": 0.001,
             "weight_decay": 0,
             "reg_type": "L2",
```

```
        "reg_lambda": 1e-4,
    },
    # 0,0003 learning rate
    {
        "label": "transformer1",
        "d_key": 64,
        "n_heads": 8,
        "mlp_factor": 4,
        "d_model": 128,
        "n_layers": 6,
        "dropout1": 0.5,
        "dropout2": 0.25,
        "optimizer_type": "Adam",
        "learning_rate": 0.0003,
        "weight_decay": 0,
        "reg_type": "L2",
        "reg_lambda": 1e-4,
    },
    # L1 reg
    {
        "label": "transformer1",
        "d_key": 64,
        "n_heads": 8,
        "mlp_factor": 4,
        "d_model": 128,
        "n_layers": 6,
        "dropout1": 0.5,
        "dropout2": 0.25,
        "optimizer_type": "Adam",
        "learning_rate": 0.001,
        "weight_decay": 0,
        "reg_type": "L1",
        "reg_lambda": 1e-4,
    },
    # More dropout
    {
        "label": "transformer1",
        "d_key": 64,
        "n_heads": 8,
        "mlp_factor": 4,
        "d_model": 128,
        "n_layers": 6,
        "dropout1": 0.6,
        "dropout2": 0.3,
        "optimizer_type": "Adam",
        "learning_rate": 0.001,
        "weight_decay": 0,
        "reg_type": "L2",
        "reg_lambda": 1e-4,
    },
]
for config in configurations:
    print(f"Training model: {config['label']}")
    model = transformer_model.TransformerClassifier(n_embeds=n_embeds, n_classes=6,␣
↪d_model=config["d_model"], d_key=config["d_key"], n_heads=config["n_heads"],␣
↪mlp_factor=config["mlp_factor"], n_layers=config["n_layers"], device = device,␣
↪dropout1=config["dropout1"], dropout2=config["dropout2"])
    results = train_model(
        label=config["label"],
        model=model,
        train_loader=train_loader,
        val_loader=val_loader,
        label_map=label_map,
        device=device,
        optimizer_type=config["optimizer_type"],
        learning_rate=config["learning_rate"],
        weight_decay=config["weight_decay"],
```

```
        reg_type=config["reg_type"],
        reg_lambda=config["reg_lambda"],
        num_epochs=num_epochs,
    )
    plot_scores(results, config["label"])
```

4 Task 4: Analysis

[ ]:
```
test_loader = DataLoader(test_dataset, batch_size=best_batch_size, shuffle=True)

predict("best_rnn_model", best_rnn_model, device, test_loader, label_map, reverse_vocab)
```

```
accuracy: 0.8245
f1: 0.8248
precision: 0.8261
recall: 0.8245
classification report:
              precision    recall  f1-score   support

     sadness       0.85      0.90      0.87       581
         joy       0.87      0.82      0.84       695
        love       0.64      0.70      0.67       159
       anger       0.83      0.81      0.82       275
        fear       0.81      0.81      0.81       224
    surprise       0.64      0.62      0.63        66

    accuracy                           0.82      2000
   macro avg       0.77      0.78      0.78      2000
weighted avg       0.83      0.82      0.82      2000

confusion matrix:
[520, 31, 4, 12, 12, 2]
[35, 571, 52, 20, 8, 9]
[16, 23, 112, 4, 2, 2]
[20, 20, 4, 223, 7, 1]
[17, 6, 2, 8, 182, 9]
[4, 7, 0, 1, 13, 41]
```

[82]:
```
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

predict("best_rnn_model", model, device, test_loader, label_map, reverse_vocab)
```

```
accuracy: 0.813
f1: 0.8115
precision: 0.8126
recall: 0.813
classification report:
              precision    recall  f1-score   support

     sadness       0.89      0.82      0.85       581
         joy       0.81      0.88      0.85       695
        love       0.68      0.59      0.63       159
       anger       0.77      0.81      0.79       275
        fear       0.80      0.80      0.80       224
    surprise       0.67      0.56      0.61        66

    accuracy                           0.81      2000
   macro avg       0.77      0.75      0.76      2000
weighted avg       0.81      0.81      0.81      2000

confusion matrix:
[479, 52, 6, 28, 13, 3]
[21, 614, 29, 15, 9, 7]
[7, 46, 94, 11, 0, 1]
[18, 19, 7, 222, 9, 0]
[15, 10, 2, 10, 180, 7]
```

```
[0, 13, 0, 3, 13, 37]
```

[ ]:
```
config = {
        "label": "transformer1",
        "d_key": 64,
        "n_heads": 8,
        "mlp_factor": 4,
        "d_model": 128,
        "n_layers": 6,
        "dropout1": 0.5,
        "dropout2": 0.25,
        "optimizer_type": "Adam",
        "learning_rate": 0.0003,
        "weight_decay": 0,
        "reg_type": "L1",
        "reg_lambda": 1e-4,
    }
model = transformer_model.TransformerClassifier(n_embeds=n_embeds, n_classes=6, d_model=config["d_model"],␣
↪d_key=config["d_key"], n_heads=config["n_heads"], mlp_factor=config["mlp_factor"],␣
↪n_layers=config["n_layers"], device = device, dropout1=config["dropout1"], dropout2=config["dropout2"])
model.load_state_dict(torch.load('models/transformer1.pth'))
predict_on_fly(model, tokenizer, vocab, device, label_map, max_length)
```

5 Task 5: Pre-trained model (transfer learning)

[ ]:
```
model_name = "distilbert-base-uncased"
num_epochs = 5
learning_rate = 5e-6
max_length = 32 # different max length as we are using raw data to be tokenized using the pretrained␣
↪tokenizer
batch_size = 32
label2id = {v: k for k, v in label_map.items()}

# Metric function
def pretrained_evaluation(predictions):
    preds = predictions.predictions.argmax(-1) # get predicted labels
    labels = predictions.label_ids
    accuracy = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds, average="weighted")
    return {"accuracy": accuracy, "f1": f1}

# Load tokenizer
tokenizer_pretrained = AutoTokenizer.from_pretrained(model_name)

# Preprocessing function
def tokenize(batch):
    return tokenizer_pretrained(batch["text"], padding="max_length", truncation=True, max_length=max_length)

# Convert pandas DataFrame to Hugging Face Dataset
train_dataset = Dataset.from_pandas(train_df)
val_dataset = Dataset.from_pandas(val_df)
test_dataset = Dataset.from_pandas(test_df)

# Tokenize datasets
train_dataset = train_dataset.map(tokenize, batched=True)
val_dataset = val_dataset.map(tokenize, batched=True)
test_dataset = test_dataset.map(tokenize, batched=True)

# Set format for PyTorch
train_dataset = train_dataset.rename_column("label", "labels")
val_dataset = val_dataset.rename_column("label", "labels")
test_dataset = test_dataset.rename_column("label", "labels")
train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
val_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
test_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
```

```python
# Load pre-trained model
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=num_classes,
    id2label=label_map,
    label2id=label2id
)

# Training arguments
training_args = TrainingArguments(
    output_dir="results",
    eval_strategy="epoch", # evaluate at the end of each epoch
    save_strategy="epoch", # save checkpoints after every epoch
    logging_strategy="epoch",
    learning_rate=learning_rate,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=64,
    weight_decay=0.01,
    num_train_epochs=num_epochs,
    load_best_model_at_end=True, # load best model at the end of training
    metric_for_best_model="f1", # specify metric to monitor
    save_total_limit=1, # keep only the best checkpoint
)

# Define Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=pretrained_evaluation, # compute metrics during evaluation
)

# Train model
trainer.train()

# Evaluate best model
best_results = trainer.evaluate()
print(f"Best Model Evaluation Results:\n {best_results}")

# Save best fine-tuned model
fine_tuned_model_name = "distilbert_finetuned"
trainer.save_model(f"models/{fine_tuned_model_name}")
tokenizer_pretrained.save_pretrained(f"models/{fine_tuned_model_name}")
```

[32]:
```python
# Load fine-tuned model
tokenizer_new = AutoTokenizer.from_pretrained(f"models/{fine_tuned_model_name}")
model_new = AutoModelForSequenceClassification.from_pretrained(f"models/{fine_tuned_model_name}",␣
↪num_labels=num_classes)
trainer = Trainer(model=model_new)

# Predict on test set
predictions = trainer.predict(test_dataset)

# Extract logits and compute predicted labels
logits = torch.tensor(predictions.predictions) # convert logits to a PyTorch tensor
predicted_labels = torch.argmax(logits, dim=1).numpy() # convert to numpy array for sklearn metrics

# Evaluate predictions
metrics = compute_metrics(test_df["label"], predicted_labels, label_map.values())
print_metrics(metrics)
save_metrics(fine_tuned_model_name, metrics)
```

```
100%|| 250/250 [00:01<00:00, 152.01it/s]
```

```
accuracy: 0.8885
f1: 0.8871
```

```
precision: 0.8869
recall: 0.8885
classification report:
              precision    recall  f1-score   support

     sadness       0.92      0.93      0.93       581
         joy       0.89      0.92      0.91       695
        love       0.74      0.70      0.72       159
       anger       0.90      0.90      0.90       275
        fear       0.90      0.87      0.88       224
    surprise       0.80      0.61      0.69        66

    accuracy                           0.89      2000
   macro avg       0.86      0.82      0.84      2000
weighted avg       0.89      0.89      0.89      2000

confusion matrix:
[543, 20, 3, 11, 4, 0]
[14, 641, 32, 6, 1, 1]
[6, 39, 111, 3, 0, 0]
[14, 7, 2, 247, 5, 0]
[11, 2, 0, 7, 195, 9]
[4, 9, 1, 0, 12, 40]
```