Code Documentation

Sentiment Analysis with Deep Learning

Technical Implementation Guide

Amrani Bouabdellah
Abdelmeraim Mohamed
Guebli Sadreddine *5th Year Statistics and Data Science*
Higher National School of Statistics and Applied Economy (ENSSEA)

Course: Introduction to Deep Learning
Instructor: Ayoub Asri
Date: February 7, 2026

Purpose of This Document

This document provides a comprehensive technical explanation of each component in the sentiment analysis project. It details what each part of the code does and identifies all tools and libraries used in the implementation. This serves as a complete technical guide for understanding the project's implementation details.

Contents

1 Data Processing Components

This section describes all components responsible for loading, cleaning, and preparing text data for model training.

1.1 Data Preprocessing (`src/data/preprocessing.py`)

Purpose: Cleans and transforms raw text data into numerical format suitable for neural network input.

Functionality:

- **Text Cleaning:** Removes URLs, HTML tags, special characters, and extra whitespace
- **Normalization:** Converts text to lowercase for consistency
- **Tokenization:** Splits text into individual words
- **Vocabulary Building:** Creates word-to-index mapping (maximum 10,000 unique words)
- **Sequence Encoding:** Converts words to numerical indices
- **Padding/Truncation:** Ensures all sequences have fixed length of 100 tokens

Tools Used:

- `nltk.tokenize` — Word-level text tokenization
- `re` (Regular Expressions) — Pattern matching for text cleaning
- `string` module — Text manipulation utilities
- Custom vocabulary dictionary — Maps words to numerical indices

Example Transformation:

```
# Input text:
"Check out this link: http://example.com!!! [emoji]"

# After preprocessing:
# 1. Cleaned: "check out this link"
# 2. Tokenized: ['check', 'out', 'this', 'link']
# 3. Encoded: [45, 123, 67, 891]
# 4. Padded to length 100: [45, 123, 67, 891, 0, 0, ..., 0]
```

Listing 1: Preprocessing Example

1.2 Data Augmentation (`src/data/augmentation.py`)

Purpose: Increases training data diversity and addresses class imbalance through text transformations.

Functionality:

- **Synonym Replacement:** Substitutes words with synonyms from WordNet
- **Random Swap:** Randomly exchanges positions of two words in the sentence
- **Random Deletion:** Removes random words with probability $p = 0.1$
- **Targeted Application:** Applies augmentation only to minority classes to balance dataset

Tools Used:

- `nltk.corpus.wordnet` — English synonym dictionary
- `random` module — Random sampling and selection
- `numpy` — Array manipulation and probability operations

Example Augmentations:

```
# Original sentence
"This product is amazing"

# Synonym replacement
"This product is fantastic"

# Random swap
"Product this is amazing"

# Random deletion
"This is amazing"
```

Listing 2: Augmentation Examples

1.3 Dataset Creation (`src/data/dataset.py`)

Purpose: Creates PyTorch-compatible dataset objects for efficient batch processing during training.

Functionality:

- **Dataset Wrapping:** Encapsulates preprocessed data in PyTorch Dataset class
- **Batch Loading:** Handles batching and shuffling during training
- **Feature Integration:** Incorporates VADER sentiment scores as additional features
- **Tensor Conversion:** Converts all data to PyTorch tensors for GPU compatibility

Tools Used:

- `torch.utils.data.Dataset` — Base dataset class for PyTorch
- `torch.utils.data.DataLoader` — Efficient batch loading with multiprocessing
- `vaderSentiment` — Rule-based sentiment feature extraction
- `torch.tensor` — Tensor creation and manipulation

2 Model Architectures

This section details all neural network architectures implemented in the project, from simple baseline models to sophisticated pretrained transformers.

2.1 Baseline Models (Built from Scratch)

2.1.1 FastText Model (`src/models/baseline/fasttext.py`)

Purpose: Implements efficient text classification using averaged word embeddings and character n-grams.

Architecture Overview:

1. **Embedding Layer:** Maps each word to 100-dimensional vector
2. **Character N-grams:** Generates subword features (3-6 character sequences)
3. **Average Pooling:** Averages all embeddings to create fixed-size representation
4. **Classification Head:** Two fully connected layers with ReLU activation

Mathematical Formulation: The final representation is computed as:

$$\mathbf{h} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{e}_{\text{word}_i} + \mathbf{e}_{\text{ngram}_i}) \tag{1}$$

where $\mathbf{e}_{\text{word}_i}$ is word embedding and $\mathbf{e}_{\text{ngram}_i}$ is character n-gram embedding.

Network Architecture:

$$\begin{aligned}\text{Input} &\to \text{Embedding}(100d) \to \text{N-gram Features}\\ &\to \text{Average Pooling} \to \text{Linear}(100 \to 128)\\ &\to \text{ReLU} \to \text{Dropout}(p = 0.3)\\ &\to \text{Linear}(128 \to 3) \to \text{Output}\end{aligned}$$

Tools Used:

- `torch.nn.Embedding` — Word embedding layer
- `torch.nn.Linear` — Fully connected layers
- `torch.nn.Dropout` — Regularization ($p = 0.3$)
- `torch.nn.ReLU` — Activation function

Parameters: Approximately 1 million trainable parameters

2.1.2 BiLSTM with Attention (`src/models/baseline/bilstm_attention.py`)

Purpose: Captures sequential dependencies in text and focuses on important words using attention mechanism.

Architecture Components:

- **Bidirectional LSTM:** Processes text in forward and backward directions (256 hidden units, 2 layers)
- **Multi-Head Attention:** Applies 4-head self-attention to identify important words
- **Feature Integration:** Concatenates LSTM output with VADER sentiment scores
- **Layer Normalization:** Stabilizes training through normalization

Network Architecture:

$$\begin{aligned}\text{Input} &\rightarrow \text{Embedding}(100d) \rightarrow \text{BiLSTM}(256h, 2\text{ layers})\\ &\rightarrow \text{MultiHeadAttention}(4\text{ heads}) \rightarrow \text{LayerNorm}\\ &\rightarrow \text{Concat}[\text{LSTM}, \text{VADER}] \rightarrow \text{Linear}(512 \rightarrow 256)\\ &\rightarrow \text{ReLU} \rightarrow \text{Dropout}(p = 0.3)\\ &\rightarrow \text{Linear}(256 \rightarrow 3) \rightarrow \text{Output}\end{aligned}$$

Attention Mechanism: The attention weights are computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

where Q, K, V are query, key, and value matrices, and d_k is the key dimension.

Tools Used:

- `torch.nn.LSTM` — Bidirectional LSTM layers
- `torch.nn.MultiheadAttention` — Self-attention mechanism
- `torch.nn.LayerNorm` — Layer normalization
- `torch.cat` — Feature concatenation

Parameters: Approximately 2.5 million trainable parameters

Key Innovation: Combines sequential processing (LSTM) with attention mechanism to focus on sentiment-bearing words.

2.1.3 Custom Transformer (`src/models/baseline/custom_transformer.py`)

Purpose: Implements transformer encoder architecture from scratch without using pre-trained weights.

Architecture Components:

- **Positional Encoding:** Adds position information using sine/cosine functions
- **Transformer Encoder:** 4 layers of self-attention and feed-forward networks
- **Multi-Head Attention:** 4 attention heads in each layer
- **Feed-Forward Network:** Two-layer MLP with expansion ($256 \rightarrow 1024 \rightarrow 256$)

Network Architecture:

$$\begin{aligned}
\text{Input} &\rightarrow \text{Embedding}(256d) + \text{Positional Encoding} \\
&\rightarrow \text{Transformer Encoder} \times 4 \text{ layers:} \\
&\quad\ \text{Multi-Head Attention}(4 \text{ heads}) \\
&\quad\ \ \rightarrow \text{Add \& Norm} \\
&\quad\ \ \rightarrow \text{Feed-Forward}(256 \rightarrow 1024 \rightarrow 256) \\
&\quad\ \ \rightarrow \text{Add \& Norm} \\
&\rightarrow \text{Global Average Pooling} \rightarrow \text{Linear}(256 \rightarrow 3)
\end{aligned}$$

Positional Encoding Formula:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \quad (3)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \quad (4)$$

where pos is position, i is dimension index, and $d_{\text{model}} = 256$.

Tools Used:

- `torch.nn.TransformerEncoder` — Transformer encoder stack
- `torch.nn.TransformerEncoderLayer` — Individual encoder layer
- Custom positional encoding implementation
- `torch.nn.LayerNorm` — Layer normalization

Parameters: Approximately 3 million trainable parameters

2.2 Pretrained Models

2.2.1 RoBERTa (`src/models/pretrained/roberta.py`)

Purpose: Fine-tunes pretrained RoBERTa (Robustly Optimized BERT) for sentiment classification.

Model Details:

- **Base Model:** `roberta-base` with 125 million pretrained parameters
- **Pretraining Data:** 160GB of text from books, Wikipedia, and web content
- **Architecture:** 12 transformer encoder layers, 768 hidden dimensions
- **Custom Head:** 3-layer classification head with layer normalization

Classification Head Architecture:

$$\begin{aligned}
\text{[CLS] token} &\to \text{Linear}(768 \to 384) \\
&\to \text{ReLU} \to \text{LayerNorm} \to \text{Dropout}(0.5) \\
&\to \text{Linear}(384 \to 192) \\
&\to \text{ReLU} \to \text{Dropout}(0.5) \\
&\to \text{Linear}(192 \to 3) \to \text{Output}
\end{aligned}$$

Training Configuration:

- Learning rate: 1×10^{-5}
- Warmup steps: 500
- Maximum sequence length: 128 tokens
- Batch size: 16
- Training epochs: 5

Tools Used:

- `transformers.RobertaModel` — Pretrained RoBERTa model
- `transformers.RobertaTokenizer` — Subword tokenization (Byte-Pair Encoding)
- `transformers.AdamW` — Optimizer with weight decay
- `transformers.get_linear_schedule_with_warmup` — Learning rate scheduler

2.2.2 BERTweet (`src/models/pretrained/bertweet.py`)

Purpose: Fine-tunes BERTweet, a BERT model specifically pretrained on Twitter data.

Model Details:

- **Base Model:** `vinai/bertweet-base` with 135 million parameters
- **Pretraining Data:** 850 million English tweets (July 2019 - November 2021)
- **Specialization:** Understands Twitter-specific language (hashtags, mentions, emojis)
- **Custom Head:** Simpler 2-layer classification head

Classification Head Architecture:

$$\begin{aligned}\text{[CLS] token} &\rightarrow \text{Linear}(768 \rightarrow 256)\\ &\rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.5)\\ &\rightarrow \text{Linear}(256 \rightarrow 3) \rightarrow \text{Output}\end{aligned}$$

Why BERTweet Performs Best:

- Pretrained on informal social media text
- Native understanding of emojis, abbreviations, and slang
- Handles noisy, ungrammatical text effectively
- Domain-specific pretraining aligns with dataset characteristics

Training Configuration:

- Learning rate: 2×10^{-5}
- Warmup steps: 500
- Maximum sequence length: 128 tokens
- Batch size: 16
- Training epochs: 4

Tools Used:

- `transformers.AutoModel` — BERTweet model loading
- `transformers.AutoTokenizer` — Twitter-aware tokenization
- Same optimization tools as RoBERTa

2.3 Ensemble Methods

2.3.1 Voting Ensemble (`src/models/ensemble/voting_ensemble.py`)

Purpose: Combines predictions from multiple models through voting mechanisms.

Voting Strategies:

1. **Soft Voting:** Averages predicted probability distributions
2. **Hard Voting:** Takes majority class vote
3. **Weighted Voting:** Weights each model by validation accuracy

Soft Voting Formula: For each class c, the final probability is:

$$P_{\text{ensemble}}(c) = \frac{1}{M}\sum_{m=1}^{M} P_m(c) \tag{5}$$

where M is the number of models and $P_m(c)$ is model m's predicted probability for class c.

Weighted Voting Formula:

$$P_{\text{weighted}}(c) = \frac{\sum_{m=1}^{M} w_m \cdot P_m(c)}{\sum_{m=1}^{M} w_m} \tag{6}$$

where w_m is the validation accuracy of model m.

Process:

```
# For each test sample:
# 1. Get predictions from all 5 models
predictions = [model1(x), model2(x), ..., model5(x)]

# 2. Average probability distributions
avg_probs = torch.mean(torch.stack(predictions), dim=0)

# 3. Select class with highest average probability
final_class = torch.argmax(avg_probs)
```

Listing 3: Soft Voting Process

Tools Used:

- `torch.stack` — Combine model outputs
- `torch.mean` — Average probabilities
- `numpy.argmax` — Final class selection

2.3.2 Stacking Ensemble (`src/models/ensemble/stacking_ensemble.py`)

Purpose: Meta-learning approach that learns optimal model combination.

Architecture:

- **Base Models:** All 5 trained models (FastText, BiLSTM, Transformer, RoBERTa, BERTweet)
- **Meta-Learner:** 3-layer neural network trained on base model predictions
- **Input Features:** Concatenated probability distributions from all base models (15 features)

Meta-Learner Architecture:

$$\begin{aligned}&[\text{Model}_1, \text{Model}_2, \ldots, \text{Model}_5] \rightarrow \text{Concatenate}(15d)\\&\rightarrow \text{Linear}(15 \rightarrow 64) \rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.3)\\&\rightarrow \text{Linear}(64 \rightarrow 32) \rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.3)\\&\rightarrow \text{Linear}(32 \rightarrow 3) \rightarrow \text{Output}\end{aligned}$$

Training Process:

1. Train all base models on training set
2. Generate predictions on validation set using trained base models
3. Train meta-learner using validation predictions as input features
4. Evaluate complete stacking ensemble on separate test set

Advantage: Meta-learner learns which model to trust for different types of samples, rather than using fixed weights.

Tools Used:

- All base model architectures
- `torch.nn` modules for meta-learner
- `torch.cat` for feature concatenation

3 Training Components

3.1 Training Loop (`src/training/trainer.py`)

Purpose: Implements complete training procedure with optimization and model checkpointing.

Functionality:

- **Forward Pass:** Computes model predictions
- **Loss Calculation:** Evaluates prediction quality
- **Backward Pass:** Computes gradients via backpropagation
- **Weight Update:** Applies optimizer step
- **Validation:** Evaluates on validation set after each epoch
- **Early Stopping:** Halts training if no improvement for n epochs
- **Checkpointing:** Saves best model based on validation accuracy

Training Algorithm:

```python
for epoch in range(num_epochs):
    model.train()
    for batch in train_loader:
        # 1. Forward pass
        predictions = model(batch.text)

        # 2. Compute loss
        loss = criterion(predictions, batch.labels)

        # 3. Backward pass
        loss.backward()

        # 4. Gradient clipping (prevent explosion)
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm
           =1.0)

        # 5. Update weights
        optimizer.step()

        # 6. Reset gradients
        optimizer.zero_grad()

    # Validation after each epoch
    val_loss, val_accuracy = validate(model, val_loader)

    # Early stopping check
    if val_accuracy > best_accuracy:
        best_accuracy = val_accuracy
        save_checkpoint(model)
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            break  # Stop training
```

Listing 4: Training Loop Pseudocode

Tools Used:

- `torch.optim.Adam` — Adaptive learning rate optimizer (baseline models)
- `torch.optim.AdamW` — Adam with weight decay (pretrained models)
- `torch.nn.utils.clip_grad_norm_` — Gradient clipping
- `torch.save` / `torch.load` — Model checkpointing
- Custom early stopping implementation

3.2 Loss Functions (`src/training/losses.py`)

Purpose: Implements custom loss functions to handle class imbalance and improve calibration.

3.2.1 Focal Loss

Motivation: Standard cross-entropy treats all samples equally. Focal loss reduces the contribution of easy examples and focuses learning on hard examples.

Formula:

$$\mathrm{FL}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t) \tag{7}$$

where:

- p_t is the model's estimated probability for the true class
- $\gamma = 2.0$ is the focusing parameter (used in this project)
- α_t is a balancing factor for class weights
- $(1 - p_t)^\gamma$ is the modulating factor that down-weights easy examples

Effect:

- When prediction is correct ($p_t \to 1$): Loss $\to 0$ (easy example)
- When prediction is wrong ($p_t \to 0$): Loss remains high (hard example)

3.2.2 Label Smoothing

Motivation: Prevents the model from becoming overconfident by softening hard labels.

Transformation:

$$y_{\text{smooth}} = (1 - \epsilon) \cdot y_{\text{true}} + \frac{\epsilon}{K} \tag{8}$$

where:

- $\epsilon = 0.1$ is the smoothing parameter
- $K = 3$ is the number of classes
- y_{true} is the one-hot encoded true label

Example:

```python
# Original hard label (positive class)
y_true = [0, 0, 1]

# After label smoothing (epsilon=0.1)
y_smooth = [0.033, 0.033, 0.933]
# Each wrong class gets small probability
# True class gets reduced probability
```

Listing 5: Label Smoothing Example

Tools Used:

- `torch.nn.functional.cross_entropy` — Base cross-entropy loss
- `torch.nn.functional.softmax` — Probability conversion
- Custom implementation of focal loss formula

3.3 Evaluation Metrics (`src/training/metrics.py`)

Purpose: Computes comprehensive performance metrics to evaluate model quality.

Metrics Implemented:

3.3.1 Accuracy

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \tag{9}$$

3.3.2 Precision

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{10}$$

3.3.3 Recall

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{11}$$

3.3.4 F1-Score

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{12}$$

3.3.5 Matthews Correlation Coefficient (MCC)

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{13}$$

MCC is particularly useful for imbalanced datasets as it considers all confusion matrix values.

Confusion Matrix: A 3×3 matrix showing true vs. predicted classes:

$$C_{ij} = \text{count of samples with true class } i \text{ predicted as class } j \tag{14}$$

Tools Used:

- `sklearn.metrics.accuracy_score`
- `sklearn.metrics.precision_recall_fscore_support`
- `sklearn.metrics.matthews_corrcoef`
- `sklearn.metrics.confusion_matrix`
- `sklearn.metrics.classification_report`

4 Evaluation Components

4.1 Model Evaluator (`src/evaluation/evaluator.py`)

Purpose: Comprehensive evaluation of trained models on test set.

Functionality:

- Loads trained model from checkpoint
- Performs inference on test set (with gradient computation disabled)
- Calculates all performance metrics
- Generates per-class performance breakdown
- Saves evaluation results in JSON and CSV formats

Evaluation Process:

```python
model.eval()  # Set to evaluation mode
with torch.no_grad():  # Disable gradient computation
    for batch in test_loader:
        # Get predictions
        outputs = model(batch.text)
        predictions = torch.argmax(outputs, dim=1)

        # Store predictions and true labels
        all_predictions.append(predictions)
        all_labels.append(batch.labels)

# Compute metrics
accuracy = compute_accuracy(all_predictions, all_labels)
f1_score = compute_f1(all_predictions, all_labels)
confusion_matrix = compute_confusion_matrix(all_predictions,
    all_labels)
```

Listing 6: Evaluation Process

Tools Used:

- `torch.no_grad()` — Disable gradient computation for efficiency
- `sklearn.metrics` — All evaluation metrics
- `pandas` — Results formatting and aggregation
- `json` — Results serialization

4.2 Error Analysis (`src/evaluation/error_analysis.py`)

Purpose: Analyzes misclassified examples to identify model weaknesses and failure patterns.

Functionality:

- **Error Identification:** Finds all incorrectly predicted samples
- **Confusion Analysis:** Categorizes errors by confusion type (e.g., Neutral → Positive)
- **High-Confidence Errors:** Identifies cases where model was confident but wrong
- **Pattern Detection:** Analyzes common characteristics of errors (text length, specific words)
- **Report Generation:** Creates detailed error reports with examples

Error Pattern Example

Neutral → Positive Errors: 234 samples
Common Patterns:

- Short affirmative phrases: “Great!”, “Thanks”
- Average prediction confidence: 0.87
- Average text length: 6 words

High-Confidence Error Example:

- Text: “Amazing and helpful”
- True Label: Neutral
- Predicted: Positive (confidence: 0.95)
- Reason: Strong positive words lacking context

Example Analysis Output:

Analysis Categories:

1. **By Confusion Type:** Which classes are most confused
2. **By Confidence:** High vs. low confidence errors
3. **By Text Length:** Short vs. long text errors
4. **By Vocabulary:** Presence of specific keywords

Tools Used:

- `numpy` — Array filtering and operations
- `pandas` — Error categorization and analysis
- Custom pattern detection functions
- Statistical analysis utilities

4.3 Visualization (`src/evaluation/visualizer.py`)

Purpose: Creates visual representations of results and model performance.

Visualizations Generated:

4.3.1 Confusion Matrices

Heatmaps showing true vs. predicted class distributions. Cell (i, j) shows the count or percentage of samples with true class i predicted as class j.

4.3.2 Training Curves

Line plots showing:

- Training loss over epochs
- Validation loss over epochs
- Training accuracy over epochs
- Validation accuracy over epochs

4.3.3 Class Distribution

Bar charts showing:

- Number of samples per class in dataset
- Class imbalance visualization
- Before and after augmentation comparison

4.3.4 Model Comparison

Bar charts comparing all models across metrics:

- Accuracy comparison
- F1-Score comparison
- Precision and Recall comparison

4.3.5 Error Distribution

Visualizations of error patterns:

- Error count by confusion type
- Confidence distribution of errors
- Text length distribution of errors

Tools Used:

- `matplotlib.pyplot` — Basic plotting framework
- `seaborn` — Statistical visualizations and heatmaps
- `numpy` — Data preparation for plotting
- `sklearn.metrics.ConfusionMatrixDisplay` — Confusion matrix visualization

Example Code:

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Create confusion matrix heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix, annot=True, fmt='d',
            cmap='Blues', xticklabels=class_names,
            yticklabels=class_names)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix - BERTweet')
plt.savefig('confusion_matrix.png', dpi=300)
```

Listing 7: Confusion Matrix Visualization

5 Utility Components

5.1 Configuration Management (`src/utils/config.py`)

Purpose: Centralized storage of all hyperparameters and configuration settings.

Configuration Categories:

5.1.1 Data Configuration

- Vocabulary size: 10,000
- Maximum sequence length: 100 tokens
- Train/validation/test split: 80% / 10% / 10%

5.1.2 Model Configuration

- Embedding dimension: 100 (baseline models)
- Hidden dimensions: 256 (LSTM, Transformer)
- Number of attention heads: 4
- Dropout rates: 0.3–0.5

5.1.3 Training Configuration

- Batch size: 32 (baseline), 16 (pretrained)
- Learning rates: 10^{-3} (baseline), 10^{-5} (pretrained)
- Number of epochs: 20–25 (baseline), 4–5 (pretrained)
- Early stopping patience: 5 (baseline), 2 (pretrained)

Example Configuration Structure:

```python
CONFIG = {
    # Data
    'vocab_size': 10000,
    'embedding_dim': 100,
    'max_length': 100,

    # Training
    'batch_size': 32,
    'learning_rate': 0.001,
    'num_epochs': 20,
    'patience': 5,

    # Model
    'hidden_dim': 256,
    'num_layers': 2,
    'dropout': 0.3,

    # System
    'device': 'cuda' if torch.cuda.is_available() else 'cpu',
    'random_seed': 42
}
```

Listing 8: Configuration Example

Tools Used:

- Python dictionaries or `dataclasses`
- `argparse` — Command-line argument parsing
- `yaml` or `json` — Configuration file loading (optional)

5.2 Logging System (`src/utils/logger.py`)

Purpose: Records training progress, hyperparameters, and errors for experiment tracking.

Functionality:

- **Event Logging:** Records all training events with timestamps
- **Hyperparameter Tracking:** Logs all configuration settings
- **Metric Recording:** Saves training and validation metrics
- **Error Logging:** Captures exceptions and error messages
- **File Management:** Creates separate log files for each experiment

Log Entry Format:

```
2026-02-06 12:23:07 - INFO - Starting training
2026-02-06 12:23:07 - INFO - Model: BiLSTM-Attention
2026-02-06 12:23:07 - INFO - Hyperparameters: {'lr': 0.003, 'batch_size': 32}
2026-02-06 12:23:45 - INFO - Epoch 1/20 - Train Loss: 0.8542, Val Acc: 0.6234
2026-02-06 12:24:23 - INFO - Epoch 2/20 - Train Loss: 0.7123, Val Acc: 0.6598
...
2026-02-06 12:45:12 - INFO - Training completed. Best Val Acc: 0.6898
2026-02-06 12:45:12 - INFO - Model saved to: results/baseline/bilstm/best_model.pt
```

Listing 9: Example Log Entry

Tools Used:

- `logging` — Python's built-in logging module
- `datetime` — Timestamp generation
- File I/O operations for log persistence

5.3 Helper Functions (`src/utils/helpers.py`)

Purpose: Common utility functions used throughout the project.

Key Functions:

5.3.1 Reproducibility

```python
def set_seed(seed=42):
    """
    Sets random seeds for reproducible results
    """
    import random
    import numpy as np
    import torch

    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

Listing 10: Seed Setting for Reproducibility

5.3.2 Device Management

```python
def get_device():
    """
    Returns appropriate device (GPU if available, else CPU)
    """
    if torch.cuda.is_available():
        device = torch.device('cuda')
        print(f'Using GPU: {torch.cuda.get_device_name(0)}')
    else:
        device = torch.device('cpu')
        print('Using CPU')
    return device
```

Listing 11: GPU/CPU Device Selection

5.3.3 Directory Management

```python
def ensure_dir(directory):
    """
    Creates directory if it doesn't exist
    """
    from pathlib import Path
    Path(directory).mkdir(parents=True, exist_ok=True)
```

Listing 12: Directory Creation

Tools Used:

- `random`, `numpy.random`, `torch.manual_seed` — Random seed setting
- `os`, `pathlib` — File system operations
- `torch.cuda` — GPU utilities and device management

5.4 Model Loading (`src/utils/model_loader.py`)

Purpose: Handles loading of saved models for inference or continued training.

Functionality:

- **Model Initialization:** Creates model architecture
- **Weight Restoration:** Loads trained weights from checkpoints
- **Version Handling:** Ensures compatibility between save and load
- **Device Transfer:** Moves model to appropriate device (CPU/GPU)

Loading Process:

```python
def load_model(model_path, model_class, device):
    """
    Loads a trained model from checkpoint
    """
    # Initialize model architecture
    model = model_class()

    # Load checkpoint
    checkpoint = torch.load(model_path, map_location=device)

    # Restore weights
    model.load_state_dict(checkpoint['model_state_dict'])

    # Move to appropriate device
    model = model.to(device)

    # Set to evaluation mode
    model.eval()

    return model
```

Listing 13: Model Loading Example

Checkpoint Structure: Saved checkpoints contain:

- `model_state_dict` — Model weights
- `optimizer_state_dict` — Optimizer state (for continued training)
- `epoch` — Training epoch number
- `best_accuracy` — Best validation accuracy achieved
- `hyperparameters` — Model configuration

Tools Used:

- `torch.load` — Load model state from disk
- `model.load_state_dict()` — Restore model weights
- `pickle` — Serialization of auxiliary objects (vocabulary, etc.)

6 Execution Scripts

This section describes the main executable scripts that orchestrate the entire pipeline.

6.1 Exploratory Data Analysis (`scripts/00_eda.py`)

Purpose: Performs comprehensive analysis of the dataset before training.

Functionality:

- Loads raw CSV data
- Computes descriptive statistics (mean length, character counts, etc.)
- Analyzes class distribution and balance
- Generates word clouds for each sentiment class
- Calculates VADER sentiment scores for comparison
- Creates visualization plots
- Saves analysis results and figures

Statistical Analysis:

- Text length distribution (words and characters)
- Class frequency counts
- Vocabulary size estimation
- Correlation between VADER scores and true labels

Visualizations Created:

- Histogram of text lengths
- Class distribution bar chart
- Word clouds (one per sentiment class)
- VADER score distributions
- Correlation heatmap

Tools Used:

- `pandas` — Data loading and statistical analysis
- `matplotlib`, `seaborn` — Visualizations
- `wordcloud` — Word cloud generation
- `vaderSentiment` — Sentiment scoring

6.2 Baseline Training (`scripts/01_train_baseline.py`)

Purpose: Trains all baseline models (FastText, BiLSTM-Attention, Custom Transformer).

Command-Line Arguments:

```
python scripts/01_train_baseline.py \
    --data_path data/raw/sentiment_dataset.csv \
    --output_dir results/baseline \
    --augmentation \
    --device cuda \
    --epochs 20 \
    --batch_size 32
```

Listing 14: Command-Line Usage

Execution Flow:

1. Parse command-line arguments
2. Load and preprocess dataset
3. Apply data augmentation if enabled
4. Create train/validation/test splits
5. For each baseline model:
   (a) Initialize model architecture
   (b) Train for specified epochs
   (c) Evaluate on validation set
   (d) Save best model checkpoint
   (e) Log training metrics
6. Generate performance summary

Tools Used:

- `argparse` — Command-line argument parsing
- All baseline model classes
- Training utilities (trainer, losses, metrics)
- Logging and checkpointing utilities

6.3 Pretrained Training (`scripts/02_train_pretrained.py`)

Purpose: Fine-tunes pretrained transformer models (RoBERTa, BERTweet).

Key Differences from Baseline Training:

- Downloads pretrained models from Hugging Face
- Uses model-specific tokenizers (subword tokenization)
- Applies lower learning rates (10^{-5} vs. 10^{-3})
- Uses AdamW optimizer with weight decay
- Implements linear warmup learning rate schedule
- Requires fewer epochs (3–5 vs. 20–25)
- Larger memory requirements (16GB+ RAM)

Training Configuration:

```python
# Optimizer with weight decay
optimizer = AdamW(model.parameters(), lr=1e-5, weight_decay=0.01)

# Learning rate scheduler with warmup
num_training_steps = len(train_loader) * num_epochs
num_warmup_steps = 500

scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=num_warmup_steps,
    num_training_steps=num_training_steps
)
```

Listing 15: Pretrained Model Training Setup

Tools Used:

- `transformers` library — Model loading and tokenization
- `torch` — Training loop implementation
- Custom evaluation functions
- GPU memory management utilities

6.4 Ensemble Training (`scripts/03_train_ensemble.py`)

Purpose: Creates and trains ensemble models combining all base models.

Execution Process:

1. Load all 5 trained base models (FastText, BiLSTM, Transformer, RoBERTa, BERTweet)
2. Generate predictions on validation set for each model
3. Train voting ensembles:
    - Hard voting
    - Soft voting (probability averaging)
    - Weighted voting (performance-based weights)
4. Train stacking ensemble:
    - Use validation predictions as features
    - Train meta-learner neural network
    - Optimize on separate meta-training set
5. Evaluate all ensemble methods on test set
6. Compare ensemble vs. individual model performance
7. Save ensemble models and results

Weight Optimization for Weighted Voting:

```
# Compute validation accuracy for each model
val_accuracies = [
    accuracy(model1_preds, val_labels),
    accuracy(model2_preds, val_labels),
    ...
]

# Normalize to create weights
weights = np.array(val_accuracies) / sum(val_accuracies)

# Apply weighted voting
weighted_probs = sum(w * p for w, p in zip(weights, all_probs))
```

Listing 16: Weight Optimization

Tools Used:

- Model loading utilities
- Ensemble model classes
- Optimization utilities for weight tuning
- Evaluation metrics

6.5 Comprehensive Evaluation (`scripts/04_evaluate_all.py`)

Purpose: Fair comparison of all models on the same test set.

Evaluation Process:

1. Load all trained models:
   - 3 baseline models
   - 2 pretrained models
   - 3 ensemble models
2. Run each model on identical test set
3. Compute all metrics for fair comparison:
   - Accuracy, Precision, Recall, F1-Score
   - Matthews Correlation Coefficient
   - Per-class performance
   - Confusion matrices
4. Generate comparison tables and visualizations
5. Perform statistical significance tests
6. Identify best-performing model
7. Create final performance report

Output Files:

- `comparison_table.csv` — Metric comparison across all models
- `performance_plots.png` — Bar charts comparing models
- `confusion_matrices.png` — All confusion matrices
- `final_report.pdf` — Comprehensive evaluation report

Tools Used:

- All evaluation utilities
- `pandas` — Results aggregation and formatting
- `matplotlib`, `seaborn` — Comparison visualizations
- Statistical testing utilities

6.6 Error Analysis (`scripts/05_error_analysis.py`)

Purpose: Detailed investigation of model errors to understand failure modes.

Analysis Components:

1. **Load Best Model:** Typically BERTweet (highest accuracy)
2. **Identify Misclassifications:** Find all incorrect predictions
3. **Categorize Errors:**
   - By confusion type (Neutral → Positive, etc.)
   - By confidence level (high vs. low confidence errors)
   - By text characteristics (length, vocabulary)
4. **Pattern Detection:**
   - Common words in errors
   - Average text length of errors
   - Presence of sarcasm indicators
5. **Generate Examples:** Show representative error cases
6. **Create Report:** Detailed error analysis document

Category	Analysis
Confusion Patterns	Most common class confusions
Confidence Analysis	High vs. low confidence errors
Length Analysis	Short vs. long text performance
Vocabulary Analysis	Problematic words/phrases
Contextual Errors	Sarcasm, irony, ambiguity

Table 1: Error Analysis Categories

Analysis Categories:

Tools Used:

- Error analysis utilities
- Statistical analysis functions
- Text pattern matching (regex)
- Natural language processing tools

7 Key Libraries Summary

Library	Version	Purpose
PyTorch	$\geq$ 2.0.0	Deep learning framework for all neural network implementations
Transformers	$\geq$ 4.30.0	Hugging Face library for pretrained models (RoBERTa, BERTweet) and tokenizers
scikit-learn	$\geq$ 1.3.0	Machine learning utilities: metrics (accuracy, F1, confusion matrix), train-test splitting
pandas	$\geq$ 2.0.0	Data manipulation, CSV loading, results aggregation and formatting
NumPy	$\geq$ 1.24.0	Numerical operations, array manipulation, mathematical computations
NLTK	$\geq$ 3.8.1	Natural language processing: tokenization, WordNet for synonyms
vaderSentiment	$\geq$ 3.3.2	Rule-based sentiment analysis for feature engineering
Matplotlib	$\geq$ 3.7.0	Basic plotting and visualization framework
Seaborn	$\geq$ 0.12.0	Statistical visualizations and heatmaps (confusion matrices)
tqdm	Latest	Progress bars for training loops
wordcloud	Latest	Word cloud generation for exploratory analysis

Table 2: Core Libraries and Their Roles

8 Reproducibility

Ensuring Reproducible Results: All experiments in this project are fully reproducible through:

1. **Fixed Random Seeds:**

```
set_seed(42)  # Applied to Python, NumPy, PyTorch
```

2. **Deterministic Algorithms:**

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

3. **Complete Hyperparameter Documentation:** All configuration settings saved in config files and logs
4. **Version-Controlled Dependencies:** `requirements.txt` specifies exact library versions
5. **Complete Training Logs:** All experiments logged in `logs/` directory with timestamps

Reproduction Steps:

```
# 1. Install exact dependencies
pip install -r requirements.txt

# 2. Download NLTK data
python -c "import nltk; nltk.download('punkt'); nltk.download('
   wordnet')"

# 3. Run complete training pipeline
python scripts/00_eda.py
python scripts/01_train_baseline.py --device cuda
python scripts/02_train_pretrained.py --device cuda
python scripts/03_train_ensemble.py --device cuda
python scripts/04_evaluate_all.py
python scripts/05_error_analysis.py

# Results will match reported values (within numerical precision)
```

Listing 17: Complete Reproduction Pipeline

9 Summary

9.1 Project Overview

This sentiment analysis project implements a comprehensive machine learning pipeline featuring:

- **3 Baseline Models:** Built from scratch (FastText, BiLSTM-Attention, Custom Transformer)
- **2 Pretrained Models:** Fine-tuned transformers (RoBERTa, BERTweet)
- **3 Ensemble Methods:** Voting (hard, soft, weighted) and stacking
- **Complete Data Pipeline:** Preprocessing, augmentation, feature engineering
- **Robust Training:** Custom loss functions, early stopping, checkpointing
- **Comprehensive Evaluation:** Multiple metrics, error analysis, visualization

9.2 Best Practices Implemented

1. **Modular Design:** Clean separation of concerns across components
2. **Reproducibility:** Fixed seeds, version control, complete logging
3. **Efficiency:** GPU acceleration, batch processing, efficient data loading
4. **Robustness:** Error handling, validation, sanity checks
5. **Documentation:** Comprehensive code documentation and technical guide

9.3 Key Technical Achievements

- Implemented transformer architecture from scratch with proper positional encoding
- Designed multi-head attention mechanism for BiLSTM model
- Applied advanced regularization (focal loss, label smoothing)
- Achieved 77.56% accuracy with BERTweet (best model)
- Created ensemble framework with meta-learning capability
- Developed comprehensive error analysis and visualization tools

9.4 Tools and Technologies

The project leverages state-of-the-art tools spanning:

- **Deep Learning:** PyTorch for neural networks, Transformers for pretrained models
- **Data Science:** NumPy, pandas, scikit-learn for data manipulation and evaluation
- **NLP:** NLTK for preprocessing, VADER for sentiment features
- **Visualization:** Matplotlib, Seaborn for plots and analysis

Conclusion

This project demonstrates a complete end-to-end implementation of a sentiment analysis system, from exploratory data analysis through model training, evaluation, and deployment considerations. The modular architecture allows for easy experimentation while maintaining code quality, reproducibility, and clarity.
The comprehensive documentation ensures that each component's functionality and purpose is clearly understood, along with the specific tools and libraries employed in the implementation.