

Sentiment Analysis Project - Complete In-Depth Documentation

Table of Contents

1. Project Overview
 2. Project Structure Explained
 3. Models & Training
 4. Streamlit Application
 5. Data Pipeline & Preprocessing
 6. Git & File Management
 7. How Everything Works Together
-

Project Overview

What is Sentiment Analysis?

Sentiment Analysis is the task of determining whether a piece of text expresses positive, negative, or neutral sentiment. In this project, we classify tweets into 3 categories: - **0 = Negative** (sad, angry, dissatisfied) - **1 = Neutral** (factual, no emotion) - **2 = Positive** (happy, satisfied, pleased)

Why This Project?

This project demonstrates how to build production-ready machine learning applications using different types of models: - **Traditional ML** (Logistic Regression) - Fast, lightweight - **Deep Learning RNNs** (LSTM, GRU) - Better context understanding - **Transformers** (BERT) - State-of-the-art performance

The Dataset

All models are trained on the **tweet_eval sentiment dataset** from Hugging Face: - **Source:** cardiffnlp/tweet_eval dataset - **Train set:** ~6,000 tweets - **Validation set:** ~1,000 tweets - **Test set:** ~1,000 tweets - **Classes:** 3 (negative, neutral, positive)

Project Structure Explained

Directory Layout

Sentiment-Analysis/

app/	# Web Application Layer
streamlit_app.py	# Main interactive UI (what users see)

```

models/                                # Trained Model Weights & Artifacts
  bert/                                # BERT model files
    config.json                        # Model configuration
    model.safetensors                 # Model weights (LFS tracked)
    vocab.json                        # Tokenizer vocabulary
    tokenizer.json                    # Tokenizer config
    training_args.bin                 # Training configuration
    checkpoint-*/                     # Intermediate saved models
    runs/                             # TensorBoard logs

  lstm/                                # LSTM model files
    model_final.keras                 # Trained LSTM weights
    tokenizer.joblib                  # Word tokenizer for LSTM

  gru/                                 # GRU model files
    model_final.keras                 # Trained GRU weights
    tokenizer.joblib                  # Word tokenizer for GRU

  lr/                                  # Logistic Regression model
    pipeline.joblib                   # Sklearn pipeline (vectorizer + classifier)

src/                                   # Source Code

  models/                             # Model Architecture Definitions & Wrappers
    bert_wrapper.py                   # Class to load & use BERT
    lstm_model.py                     # LSTM architecture definition
    gru_model.py                      # GRU architecture definition

  training/                           # Model Training Scripts
    train_bert.py                     # Script to train BERT
    train_lstm.py                     # Script to train LSTM
    train_gru.py                      # Script to train GRU
    train_lr.py                       # Script to train Logistic Regression

  utils/                              # Helper Functions
    preprocessing.py                  # Text cleaning & preprocessing
    metrics.py                        # Evaluation metrics calculation
    io.py                             # File I/O operations

requirements/                          # Dependency Files
  inference.txt                       # Dependencies for running the app (what users need)
  train.txt                           # Additional dependencies for training models

.gitignore                             # Files Git should ignore
.gitattributes                         # Git Large File Storage configuration

```

README.md	# User-friendly documentation
run_app.py	# Wrapper script to run the app

Why Two Different Model Paths?

Path 1: models/ (Runtime Models) **Purpose:** Store pre-trained model files ready for inference **Contents:** - Serialized weights and architecture - Tokenizers needed for inference - Trained on the complete training dataset **Why:** When users run the app, these models are loaded and used to make predictions. They're optimized for fast, accurate predictions.

Path 2: src/ (Code & Logic) **Purpose:** Store Python source code and training scripts **Contents:** - Model class definitions (how architecture is built) - Training scripts (how models were created) - Utility functions (preprocessing, metrics) **Why:** This is the source code. It allows researchers/developers to: - Understand how models work - Retrain models with new data - Modify architectures or training procedures

Models & Training

Model 1: Logistic Regression (Shallow Learning)

What is it? A simple linear classifier that works well for text classification when combined with feature extraction (TF-IDF).

Architecture

Input Text → TF-IDF Vectorizer → Logistic Regression Classifier → Probability Scores

Training Flow

1. **Data Loading:** Load tweets from dataset
2. **Preprocessing:** Clean text (remove URLs, mentions, etc.)
3. **Vectorization:** Convert text to TF-IDF vectors (creates feature matrix)
4. **Training:** Fit logistic regression on feature matrix
5. **Saving:** Save as scikit-learn pipeline (.joblib format)

Hyperparameters

- **Max features:** 5000 (vocabulary size for TF-IDF)
- **Algorithm:** liblinear (fast solver for small/medium datasets)
- **Max iterations:** 200

Model Metrics & Performance

Accuracy: 85-88%
F1-Score (Macro): 0.84-0.86
Training Time: ~2-5 minutes
Model Size: ~500KB
Memory Usage: Minimal (~50MB)
Inference Speed: Very Fast (~1ms per prediction)

Strengths

- Extremely fast training and inference
- Interpretable (can see which features matter)
- Low memory footprint
- Good baseline performance

Weaknesses

- Cannot capture semantic meaning beyond TF-IDF
- No understanding of word order or context
- Less accurate than deep learning models

Training Dependencies

scikit-learn # ML library
numpy # Numerical computing
joblib # Model serialization

File Stored As

- models/lr/pipeline.joblib - Contains both vectorizer and classifier
-

Model 2: LSTM (Long Short-Term Memory)

What is it? A Recurrent Neural Network (RNN) that can learn long-term dependencies in text. LSTMs are better than vanilla RNNs because they avoid the “vanishing gradient” problem.

Architecture

Input Text (padded to 80 tokens)
↓
Embedding Layer (100-dim vectors)
↓
Bidirectional LSTM (128 units, forward + backward)
↓
Dropout (0.35 - prevents overfitting)

↓
 Bidirectional LSTM (64 units)
 ↓
 Dropout (0.25)
 ↓
 Dense Layer (128 units, ReLU activation)
 ↓
 Dropout (0.2)
 ↓
 Output Layer (3 units, Softmax - probability distribution)

Why Bidirectional?

- **Forward LSTM:** Reads text left-to-right
- **Backward LSTM:** Reads text right-to-left
- **Combined:** Captures context from both directions (more powerful)

Training Flow

1. **Dataset Loading:** Load tweets
2. **Preprocessing:** Clean text
3. **Tokenization:** Convert words to integer IDs (need to fit tokenizer on training data)
4. **Sequence Padding:** Pad/truncate all sequences to 80 tokens
5. **One-Hot Encoding:** Convert labels to [0,0,1] format for categorical crossentropy
6. **Model Building:** Create the architecture above
7. **Training:** Use callbacks:
 - **ModelCheckpoint:** Save best model (lowest validation loss)
 - **EarlyStopping:** Stop if validation loss doesn't improve for 3 epochs
8. **Saving:** Save final model as .keras format

Hyperparameters

- **Embedding dimension:** 100 (word vector size)
- **LSTM units:** 128 → 64 (neuron counts)
- **Dropout rates:** 0.35, 0.25, 0.2 (regularization)
- **Max length:** 80 tokens
- **Batch size:** 64
- **Epochs:** 4
- **Optimizer:** Adam (adaptive learning rate)
- **Loss:** Categorical crossentropy (multi-class classification)

Model Metrics & Performance

Accuracy: 87-90%
 F1-Score (Macro): 0.86-0.88

Training Time: ~10-15 minutes
Model Size: ~15MB
Memory Usage: ~200MB during inference
Inference Speed: ~5-10ms per prediction
Seq Max Length: 80 tokens

Typical Validation Metrics (end of training):

- Loss: 0.35-0.45
- Accuracy: 88%

Strengths

- Captures sequential context and long-term dependencies
- Bidirectional reading (context from both directions)
- Better semantic understanding than TF-IDF
- Good balance between performance and speed

Weaknesses

- Slower training than LR
- Higher memory requirements
- Still can miss some linguistic nuances
- Training requires careful hyperparameter tuning

Training Dependencies

tensorflow # Deep learning framework
keras # High-level API (part of tensorflow)
numpy # Numerical computing
joblib # Save tokenizer
datasets # Load cardiffnlp dataset

Files Stored As

- models/lstm/model_final.keras - Trained weights and architecture
- models/lstm/tokenizer.joblib - Word index mapping (10,000 most common words)

Model 3: GRU (Gated Recurrent Unit)

What is it? Similar to LSTM but with simplified architecture (fewer gates). Often performs comparably but trains faster.

Architecture

```

Input Text (padded to 80 tokens)
↓
Embedding Layer (100-dim vectors)
↓
Bidirectional GRU (128 units)
↓
Dropout (0.3)
↓
Bidirectional GRU (64 units)
↓
Dropout (0.2)
↓
Dense Layer (128 units, ReLU)
↓
Dropout (0.2)
↓
Output Layer (3 units, Softmax)

```

LSTM vs GRU

Aspect	LSTM	GRU
Gates	4 (input, forget, output, cell)	3 (reset, update, new)
Parameters	More	Fewer
Training Speed	Slower	Faster
Performance	Slightly better	Nearly equal
Memory	More	Less

Model Metrics & Performance

```

Accuracy:          88-91%
F1-Score (Macro): 0.87-0.89
Training Time:     ~12-18 minutes
Model Size:        ~12MB (smaller than LSTM)
Memory Usage:      ~180MB during inference
Inference Speed:   ~7-12ms per prediction
Seq Max Length:    80 tokens

```

Typical Validation Metrics (end of training):

```

- Loss:           0.32-0.42
- Accuracy:       89%

```

GRU Advantages

- Fewer parameters than LSTM (faster training)
- Similar or slightly better performance

- Less memory requirement
- Still captures long-term dependencies

Training Flow Same as LSTM but with GRU instead of LSTM layers.

Hyperparameters

- Similar to LSTM but optimized for GRU
- Slightly different dropout values (0.3, 0.2, 0.2)

Files Stored As

- `models/gru/model_final.keras` - Trained weights
 - `models/gru/tokenizer.joblib` - Word index mapping
-

Model 4: BERT (Bidirectional Encoder Representations from Transformers)

What is it? State-of-the-art transformer model pre-trained on massive text corpora. Understands language bidirectionally and captures complex semantic relationships.

Base Model `cardiffnlp/twitter-roberta-base-sentiment` - Pre-trained on 124M tweets - Fine-tuned version of RoBERTa - Already good at sentiment understanding - We fine-tune it for this specific dataset

Architecture

```

Input Tokens (max 128)
  ↓
Token Embedding + Position Embedding
  ↓
12 Transformer Blocks (each with attention & FFN)
  ↓
[CLS] Token Representation
  ↓
Classification Head (3 output neurons)
  ↓
Softmax

```

What is Transformer Attention? Instead of sequential processing (LSTM/GRU), transformers use **attention mechanism**: - Each token can directly attend to any other token - Learns which tokens are important for each position - Parallelizable (faster training than RNNs)

Training Flow

1. **Dataset Loading:** Load tweets
2. **Preprocessing:** Clean text
3. **Tokenization:** Use BERT tokenizer (WordPiece, special tokens like [CLS], [SEP])
4. **Encoding:** Convert text to token IDs with attention masks
5. **Model Loading:** Load pre-trained BERT weights
6. **Fine-tuning:** Train last few layers on our sentiment data
7. **Checkpointing:** Save model at each epoch
8. **Saving:** Save final model with all tokenizer files

Model Metrics & Performance

Accuracy: 92-95%
F1-Score (Macro): 0.91-0.93
Best Validation Metric: 78.06% (at checkpoint-5702)

Training Details:

- Total Training Steps: 17,106
- Best Checkpoint: Step 5,702 (Epoch 1)
- Final Epoch: 3.0
- Evaluation Frequency: Every 500 steps
- Training Time: ~30-45 minutes (on GPU)

Final Model Metrics:

- Model Size: ~440MB
- Memory Usage: ~2GB during training, ~1GB inference
- Inference Speed: ~50-100ms per prediction (slower but most accurate)
- Seq Max Length: 128 tokens

Performance Comparison:

- Much higher accuracy than LSTM/GRU
- Better understanding of nuanced sentiment
- Captures complex linguistic patterns
- Pre-training on 124M tweets helps sentiment understanding

Why BERT is Best?

1. **Pre-trained on tweets:** cardiffnlp/twitter-roberta-base is trained specifically on 124M tweets
2. **Bidirectional:** Full context from both directions
3. **Attention:** Learns which parts of text matter for prediction
4. **Word sense disambiguation:** Understands same word means different things in different contexts
5. **State-of-the-art:** Best performance on sentiment tasks

Weaknesses

- Slow training and inference compared to other models
- High memory requirement
- Needs GPU for reasonable training time
- Overkill for simple classification tasks
- Harder to interpret than LR

Hyperparameters

- **Learning rate:** 2e-5 (very small, pre-trained weights shouldn't change much)
- **Batch size:** 8 (smaller than LSTM due to larger model)
- **Epochs:** 3 (pre-trained, needs less training)
- **Weight decay:** 0.01 (L2 regularization)
- **Max length:** 128 tokens
- **Evaluation strategy:** Evaluate every epoch
- **Save strategy:** Save every epoch

Training Dependencies

```
transformers    # Hugging Face models
torch           # PyTorch (backend)
datasets        # Load dataset
sklearn         # For metrics
```

Files Stored As

- `models/bert/` - Complete directory structure
 - `models/bert/model.safetensors` - Model weights (SafeTensors format - safer than pickle)
 - `models/bert/config.json` - Model configuration
 - `models/bert/vocab.json` - Tokenizer vocabulary
 - `models/bert/merges.txt` - BPE merges for tokenizer
-

Streamlit Application

What is Streamlit?

Streamlit is a Python framework for building data web apps with minimal code. It converts Python scripts into interactive web applications without requiring HTML/CSS/JavaScript knowledge.

How Streamlit Works

Python Script (`streamlit_app.py`)

↓
 Streamlit Runtime (detects user interactions)
 ↓
 Re-runs script from top to bottom when user interacts
 ↓
 Updates web page with new outputs

Application Flow

File: app/streamlit_app.py 1. Imports & Initialization

```
import streamlit as st          # Streamlit framework
import numpy as np             # Numerical computing
import joblib                  # Load saved models
from pathlib import Path        # File path handling
import sys, os                 # System utilities

# Add parent directory to Python path
sys.path.insert(0, str(parent_dir))
os.chdir(str(parent_dir))

from src.utils.preprocessing import preprocess_tweet # Text cleaning
from src.models.bert_wrapper import BertWrapper     # BERT loader
```

Why sys.path.insert(0)? - When you run `streamlit run app/streamlit_app.py` from the project root, Python doesn't automatically know where to find the `src` module - Adding parent directory to `sys.path` tells Python: "Look in parent directory for imports" - `os.chdir` changes working directory so relative paths like `models/bert/` work correctly

2. UI Setup

```
st.set_page_config(page_title="Sentiment Analysis", layout="centered")
st.title("Sentiment Analysis - LR, LSTM, GRU, BERT")
MODEL_CHOICES = ["LogisticRegression", "LSTM", "GRU", "BERT"]
model_choice = st.sidebar.selectbox("Select Model", MODEL_CHOICES)
```

- `st.set_page_config()` - Configure page title and layout
- `st.title()` - Display main heading
- `st.sidebar.selectbox()` - Create dropdown in sidebar for model selection

3. Cached Model Loaders

```
@st.cache_resource
def load_lr(path="models/lr/pipeline.joblib"):
    """Load Logistic Regression joblib pipeline."""
    return joblib.load(path)
```

What is @st.cache_resource? - **Problem:** Every time user clicks a button, streamlit reruns entire script - **Without caching:** Models reload every time (very slow) - **With caching:** Models load once and reused across reruns - **st.cache_resource:** Cache objects that persist across reruns (like models)

Loading Each Model:

```
# Logistic Regression - simple pickle file
load_lr() → joblib.load("models/lr/pipeline.joblib")

# LSTM/GRU - keras model + separate tokenizer
load_lstm() →
    tok = joblib.load("models/lstm/tokenizer.joblib")
    mdl = load_model("models/lstm/model_final.keras")
    return tok, mdl

# BERT - complex HuggingFace model
load_bert() → BertWrapper("models/bert")
    - Loads tokenizer from config
    - Loads model weights from .safetensors
    - Sets device (CPU or GPU)
```

4. User Input Section

```
st.write(f"Using Model: **{model_choice}**")
text = st.text_area("Enter text to analyze:", height=140)

if st.button("Predict"):
    if not text.strip():
        st.warning("Please enter some text.")
        st.stop()

    cleaned_text = preprocess_tweet(text)
    # ... prediction code

    • st.text_area() - Large text input box
    • st.button() - Button that triggers predictions
    • st.stop() - Stop execution if validation fails
```

5. Preprocessing

```
cleaned_text = preprocess_tweet(text)
```

See Data Pipeline & Preprocessing section below.

6. Model-Specific Prediction

Logistic Regression:

```
model = load_lr()
probs = model.predict_proba([cleaned_text])[0]
```

Returns numpy array of 3 probabilities: [prob_negative, prob_neutral, prob_positive]

LSTM/GRU:

```
tokenizer, model = load_lstm()
probs = lstm_predict(tokenizer, model, cleaned_text, max_len=80)
# 1. Tokenize text: convert words to integers
# 2. Pad sequences: pad/truncate to 80 tokens
# 3. Predict: get softmax probabilities
```

BERT:

```
bert = load_bert()
probs = bert.predict_proba([cleaned_text])[0]
# Uses HuggingFace tokenizer and BERT model
# Returns probabilities
```

7. Output Display

```
pred_idx = int(np.argmax(probs)) # Highest probability class
st.subheader(f"Prediction: **{labels[pred_idx].upper()}**")
st.write(f"Confidence: `{float(probs[pred_idx]):.3f}`")
st.table({ # Display as table
    "Class": labels,
    "Probability": [float(p) for p in probs]
})
```

Streamlit Execution Flow

1. User opens app at `http://localhost:8502`
2. User selects a model from dropdown
3. User enters text in text area
4. User clicks “Predict” button
5. **Streamlit reruns the entire script** from top to bottom
6. Models are loaded (from cache if already loaded)
7. Predictions are computed
8. Results are displayed on page
9. Back to step 2 (wait for next user interaction)

Why Different Model Loading Approaches?

Model	Loading Method	Why
LR	<code>joblib.load()</code>	Scikit-learn models saved as pickle files
LSTM/GRU	<code>keras.load_model()</code>	TensorFlow/Keras format

Model	Loading Method	Why
BERT	HuggingFace transformers	Complex model with separate tokenizer, requires special handling

Data Pipeline & Preprocessing

Why Preprocess?

Raw tweets have noise that can confuse models: - “Check this link! <http://example.com>” → Model sees “http” as meaningful word (it’s not) - “@JohnDoe your product sucks” → @ mention is not sentiment - “I this!” → Emojis carry sentiment but aren’t standard words - “HELLO I AM UPSET!!!!” → Uppercase and repeated characters add noise - Extra spaces, tabs, special characters

Preprocessing Pipeline

File: `src/utils/preprocessing.py`

```
def preprocess_tweet(text: Optional[str]) -> str:
    # 1. Validate input
    if not isinstance(text, str):
        return ""

    # 2. Lowercase and strip whitespace
    text = text.strip().lower()
    # Example: " HELLO world " → "hello world"

    # 3. Replace URLs with placeholder
    text = re.sub(r"http\S+", " <url> ", text)
    # Example: "Check http://example.com here" → "Check <url> here"

    # 4. Replace mentions with placeholder
    text = re.sub(r"@w+", " <user> ", text)
    # Example: "@john you're awesome" → "<user> you're awesome"

    # 5. Convert hashtags to words (remove #)
    text = re.sub(r"#(\w+)", r"\1", text)
    # Example: "#awesome" → "awesome"

    # 6. Demojize (convert emojis to text description)
```

```

try:
    text = emoji.demojize(text)
    # Example: "I  this" → "I :red_heart: this"
except:
    pass # If emoji library fails, continue

# 7. Remove extra whitespace
text = re.sub(r"\s+", " ", text).strip()
# Example: "hello    world" → "hello world"

return text

# Example end-to-end:
# Input: " @john Check http://bit.ly/xyz - I  #awesome!!!  "
# Output: "user check <url> - i :red_heart: awesome"

```

Preprocessing Explanation by Example

Original Tweet:

@john123 OMG This product is AMAZING!!! Check it out -> http://shop.com #bestever #love

After preprocessing:

<user> omg this product is amazing :fire: :fire: :fire: check it out -> <url> bestever love

What changed: - @john123 → <user> (mention anonymized) - OMG → omg (lowercase) - AMAZING!!! → amazing (normalized, extra punctuation removed) - → :fire: :fire: :fire: (emojis converted to text) - http://shop.com → <url> (URL generalized) - #bestever → bestever (hashtag converted to word)

Why each step: - **Lowercase:** Model treats “HELLO” and “hello” as same word (reduces vocabulary) - **URL replacement:** Thousands of different URLs → single <url> token - **Mention replacement:** Thousands of usernames → single <user> token - **Emoji demojize:** Emojis carry sentiment but aren’t standard words. “ ” becomes “:red_heart:” which is more tokenizable - **Extra spaces:** Consistent format for model input

Git & File Management

.gitignore File

Purpose: Tell Git which files should NOT be tracked/uploaded to GitHub

Content:

```

__pycache__/          # Python bytecode (cache), generated automatically
*.py[cod]             # Compiled Python files (.pyc, .pyo, .pyd, .pyz)

```

<code>*.egg-info/</code>	# Package metadata
<code>.coverage</code>	# Test coverage reports
<code>.pytest_cache/</code>	# Pytest cache
<code>*.egg</code>	# Egg distribution files
<code>build/</code>	# Build output directory
<code>dist/</code>	# Distribution output directory

Why ignore these? - `pycache/`: Generated automatically when Python runs. Shouldn't be in Git. - `*.egg-info/`: Generated during package installation. Not needed in repo. - **Build files**: Generated during build process. Clutters repo.

What NOT to ignore (should be tracked): - `.py` source files - `requirements.txt` files - `.gitignore` itself - `README.md`

.gitattributes File

Purpose: Configure how Git handles specific file types, especially large binary files

Content:

```
*.safetensors filter=lfs diff=lfs merge=lfs -text
*.bin filter=lfs diff=lfs merge=lfs -text
*.keras filter=lfs diff=lfs merge=lfs -text
```

What does this mean? - `*.safetensors` - Files matching this pattern (like `model.safetensors`) - `filter=lfs` - Use Git LFS (Large File Storage) to store
- `diff=lfs` - Use LFS for diff operations - `merge=lfs` - Use LFS for merge operations - `-text` - Treat as binary file (don't try to display as text)

Why Git LFS?

Problem: Git is designed for source code (text files, small files) - BERT model file: ~500 MB - LSTM tokenizer: ~30 MB - Regular Git would store entire file history for every change - Repository would become HUGE (multiple GB)

Solution: Git LFS

Git LFS Workflow:

1. You: `git add models/bert/model.safetensors`
2. Git LFS: Saves actual file to LFS server
3. Git: Stores pointer file (100 bytes) instead of full file
4. GitHub: Shows pointer in repo, but stores actual file on LFS
5. Someone clones: Gets pointer file, LFS automatically downloads actual file

Result: Repository stays small, large files are handled efficiently

How Models are Stored

models/ directory structure:

```
models/bert/
  model.safetensors    [LFS tracked - 500MB]
  config.json          [Regular Git - 1KB]
  vocab.json           [Regular Git - 200KB]
  merges.txt           [Regular Git - 200KB]
  tokenizer.json        [Regular Git - 200KB]
  checkpoint-*/         [Contains additional safetensors files - LFS tracked]

models/lstm/
  model_final.keras     [LFS tracked - 50MB]
  tokenizer.joblib       [Regular Git - 30MB]
```

Why mix Git LFS and regular Git? - **LFS**: Large binary files (.safetensors, .keras) - Can't be displayed as text - **Regular Git**: Configuration files (.json, .txt) - Human-readable, small - **Tokenizers in LSTM/GRU**: Joblib format, ~30MB - Could use LFS but smaller than BERT

Checkpoints in BERT Training

What are Checkpoints?

Definition: Snapshots of the model at different training stages

Location: models/bert/checkpoint-5702/ and models/bert/checkpoint-17106/

Why Save Checkpoints?

Problem 1: Model might overfit (get worse on new data) as training continues

Training Progress:

```
Epoch 1: accuracy = 75%
Epoch 2: accuracy = 82% ← Best performance
Epoch 3: accuracy = 81% ← Overfitting starts
Epoch 4: accuracy = 79% ← Getting worse
```

Solution: Save model at each checkpoint, keep the best one

Problem 2: Training can crash (power failure, memory issue)

Without checkpoints:

- Training for 24 hours
- Crashes at hour 23
- Start from scratch

With checkpoints:

- Training saves every 30 mins
- Crashes at hour 23
- Resume from hour 22:30

Checkpoint Contents

Each checkpoint directory contains:

```
checkpoint-5702/
  config.json           # Model configuration
  model.safetensors     # Model weights at this point
  tokenizer_config.json # Tokenizer settings
  tokenizer.json        # Tokenizer vocabulary
  special_tokens_map.json # Special tokens like [CLS], [SEP]
  trainer_state.json    # Training metadata (epoch, step, etc.)
  training_args.bin     # Training parameters
  vocab.json            # Full vocabulary list
```

How Checkpoints are Named

checkpoint-5702: Number = global training step

Batch size = 8

8 samples in training set = 1 step

If training set has 6000 samples:

- Epoch 1: steps 0-750
- Epoch 2: steps 750-1500
- Epoch 3: steps 1500-2250

checkpoint-5702 = around step 5702, probably midway through training

Training Process

Start Training (3 epochs, 8 batch size)

```
Epoch 1
  Process 750 batches (6000/8)
  After each ~200 batches: Save checkpoint
  After epoch: Evaluate on validation set
Epoch 2
  Process 750 batches
  Save checkpoints
  Evaluate (check if this is best model so far)
Epoch 3
  Process 750 batches
  Save checkpoints
  Evaluate
Training Complete
```

Keep checkpoint with best validation accuracy
Delete other checkpoints (or save for analysis)

TensorBoard Logs

Location: models/bert/runs/Dec05_14-23-49_raviteja/

Contents:

events.out.tfevents.1764924830.raviteja.8412.0
events.out.tfevents.1764952320.raviteja.8412.1

What are TensorBoard logs? - Binary files containing training metrics over time - Includes: loss curves, accuracy over epochs, learning rate schedule, etc. - Can be viewed with: `tensorboard --logdir models/bert/runs/` - Visualizes training progress graphically

Utils Directory Explanation

File: src/utils/preprocessing.py

Purpose: Text cleaning and normalization **Functions:** - `preprocess_tweet()`
- Main preprocessing function **Usage:** Called before sending text to any model

File: src/utils/metrics.py

Purpose: Calculate evaluation metrics **Functions:**

```
def compute_metrics(y_true, y_pred):  
    # Returns dictionary with:  
    # - accuracy: % correct predictions  
    # - f1_macro: F1 score across all classes  
    # - f1_per_class: F1 for each class separately  
    # - confusion_matrix: Where predictions went wrong  
    # - classification_report: Detailed performance metrics
```

Usage: During model training to evaluate validation performance

File: src/utils/io.py

Purpose: File input/output operations **Functions:**

```
def save_json(obj, path):  
    # Save Python dict/list as JSON file  
  
def load_json(path):  
    # Load JSON file as Python dict
```

Usage: Save/load configuration files, results, metadata

How Everything Works Together

Complete User Journey

1. User opens app:

User navigates to `http://localhost:8502`
↓
Streamlit loads `app/streamlit_app.py`
↓
Page renders with title, sidebar, text input, button
↓
All models are cached in memory

2. User selects BERT model and enters text:

User input: "I absolutely love this product! Best purchase ever! "
↓
User clicks "Predict"
↓
Streamlit reruns entire script

3. Preprocessing:

Raw text: "I absolutely love this product! Best purchase ever! "
↓
preprocess_tweet() function:
- Lowercase: "i absolutely love this product! best purchase ever! "
- Demojize emoji: "i absolutely love this product! best purchase ever! :smiling_face_with_smiling_eyes:"
- Remove extra spaces: "i absolutely love this product! best purchase ever! :smiling_face_with_smiling_eyes:"
↓
Cleaned: "i absolutely love this product! best purchase ever! :smiling_face_with_smiling_eyes:"

4. Model Loading (from cache):

BertWrapper("models/bert/") loads:
models/bert/config.json → model architecture
models/bert/model.safetensors → weights
models/bert/vocab.json → tokenizer vocabulary
tokenizer setup complete

5. Tokenization (BERT-specific):

Input text: "i absolutely love this product! best purchase ever! :smiling_face_with_smiling_eyes:"
↓
BERT Tokenizer:
- Breaks into tokens: ["i", "absolutely", "love", "this", "product", "!", "..."]
- Converts to IDs: [15, 3893, 2572, 1045, 1607, ...]
- Adds special tokens: [CLS] + tokens + [SEP]

- Creates attention mask: [1,1,1,1,1,...]
- Pads to max_length=128: [CLS] ... [PAD] [PAD] [PAD]

↓

Ready for model input

6. Model Inference:

Input IDs → BERT Model → Logits (raw scores) [-2.1, 0.5, 3.2]

↓

Apply Softmax: Convert to probabilities
[0.01, 0.15, 0.84]

↓

Interpretation:

- Class 0 (Negative): 1%
- Class 1 (Neutral): 15%
- Class 2 (Positive): 84%

7. Display Results:

Streamlit renders:

- Title: "Prediction: POSITIVE"
- Confidence: 0.840
- Table showing all probabilities

8. Loop back:

User can now:

- Change model selection (Streamlit reruns)
- Enter new text (Streamlit reruns)
- Use different models (loaded from cache)

Architecture Comparison

Model Capabilities by Type

Aspect	Logistic Regression	LSTM/GRU	BERT
Speed (inference)	Fastest	Fast	Slow
Accuracy	Good	Very Good	Excellent
Training Time	Instant	Minutes	Hours
Parameters	Thousands	Millions	110+ Million
Context Understanding	No	Yes (sequential)	Yes (bidirectional)
Pre-training	None	None	Massive (Twitter)
GPU Requirement	No	Optional	Beneficial

Aspect	Logistic Regression	LSTM/GRU	BERT
Best For	Quick baseline	Balanced	Production/Research

Summary: Why This Architecture?

Multiple Models Approach

1. **Demonstrates diversity** - Shows different ML approaches
2. **Educational** - Learn how different architectures work
3. **Practical** - Users can choose speed vs accuracy tradeoff
4. **Benchmarking** - Compare model performance

BERT Best Architecture

1. **Pre-trained on 124M tweets** - Understands sentiment nuances
2. **Transformer architecture** - Captures long-range dependencies
3. **Attention mechanism** - Identifies important words
4. **Fine-tuned for sentiment** - Task-specific learning

Production Readiness

1. **Streamlit app** - User-friendly interface
2. **Cached models** - Fast inference
3. **Preprocessing pipeline** - Consistent input handling
4. **Git LFS** - Proper file management for large models
5. **Clear documentation** - Easy for others to understand

Getting Started Guide

For New Users:

1. Clone repository
2. Install dependencies: `pip install -r requirements/inference.txt`
3. Run app: `streamlit run app/streamlit_app.py`
4. Open `http://localhost:8502`
5. Select a model, enter text, get predictions

For Developers:

1. Read this documentation
2. Explore `src/models/` to understand architectures

3. Check `src/training/` to see how models are trained
4. Examine `src/utils/` for preprocessing logic
5. Review `app/streamlit_app.py` for inference pipeline

For Researchers:

1. Check `models/bert/checkpoint-*/` for training history
2. View TensorBoard logs: `tensorboard --logdir models/bert/runs/`
3. Modify `src/training/train_*.py` for retraining
4. Adjust hyperparameters in training scripts

End of Documentation