Maintenance Guide

Authorship Verification using Impostor Projections and Siamese Networks

# Objective

Ensure seamless maintainability and extensibility of the Siamese BERT-based Authorship Verification System, developed with Object-Oriented Programming (OOP) principles for scalability, modularity, and testability.

# 1. Software Architecture

This project was intentionally written using OOP, not functional programming. We knew we had to:  
- Carefully manage inter-module dependencies  
- Minimize error propagation in long pipeline stages  
- Enforce clear modular boundaries (preprocessing, training, inference, logging)  
- Prevent shared-state bugs and global state misuse common in functional pipelines  
  
Each class encapsulates state and functionality (e.g., Trainer, Procedure, SignalGeneration), with single-responsibility design. This made debugging and tracing errors across a lengthy multi-step NLP pipeline more reliable.

# 2. Directory Structure

Here is the updated directory structure from the actual project (README.md):

bert\_siamese\_authorship\_verification/

│

├── config/

│ └── config.yaml # Central configuration file for paths, models, hyperparameters

│

├── data/ # Main data folder

│ ├── raw/ # Unprocessed raw `.txt` files (Shakespeare and impostors)

│ ├── processed/ # Processed texts stored as JSONs in various data structures for usage in the procedures

│

├── plots/ # Auto-saved figures (t-SNE, signals) used in local analysis and WandB

│

├── saved\_trained\_models/ # Stores all model artifacts

│ ├── berts/ # Fine-tuned BERTs (one per impostor author)

│ └── models/ # Siamese networks (one per impostor pair)

│

├── src/ # Core logic and pipeline engine

│ ├── model.py # Siamese CNN-BiLSTM network using BERT embeddings

│ ├── trainer.py # Handles training loop, evaluation, callbacks

│ ├── procedure.py # High-level orchestration singleton, entry to all pipeline steps

│ ├── data\_loader.py # Loads pairs, training chunks, evaluation data

│ ├── preprocess.py # Preprocessing logic (cleaning, tokenization, chunking)

│ ├── signal\_generation.py # Signal creation from Siamese model output

│ ├── distance\_manager.py # Manages DTW matrix calculation between signals

│ ├── isolation\_forest.py # Trains IsolationForest models for anomaly scoring

│ ├── clustering.py # K-Medoids clustering on anomaly score matrix

│

├── utilities/ # Shared utilities used throughout the pipeline

│ ├── bert\_fine\_tuner.py # MLM fine-tuning of BERT per impostor

│ ├── config\_loader.py # Loads and validates YAML configuration

│ ├── dataset\_manager.py # Constructs training JSON datasets from raw text

│ ├── data\_visualizer.py # t-SNE visualizer and signal plotter (WandB integrated)

│ ├── increment\_last\_iteration.py # Manages iteration count for saving new runs

│ ├── load\_json\_data.py # Safely reads JSON into Python objects

│ ├── logger.py # Logs structured messages and events

│ ├── make\_pairs.py # Constructs impostor pair combinations for training

│ ├── wandb\_helpers.py # Sets run names, uploads artifacts to WandB

│ ├── \_\_init\_\_.py # Package initializer

│

├── main.py # Interactive command-line controller, allows user to execute system modules

├── requirements.txt # Lists all required packages (TF, HuggingFace, WandB, etc.)

└── README.md # Project overview and high-level documentation

# 3. Environment Setup

TensorFlow-GPU 2.10.0 does not include Keras, and must be installed separately (e.g., keras==2.10.0). It was used because it was compatible with our GPUs and CUDA versions. Regular Tensorflow of this older minor version, does not support GPU runtime.  
TensorFlow 2.18.0 on Colab includes Keras 3, which is currently incompatible with transformers 4.51.0. To fix this, we use:  
  
os.environ["TF\_USE\_LEGACY\_KERAS"] = "1"  
  
This allows transformers to work with Keras again on Colab.

To run locally with Tensorflow-GPU 2.10.0, we must ensure the current CuDNN and CUDA versions. Follow the steps to ensure compatibility:

1. Download CuDNN 11.2: <https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.1.1.33/11.2_20210301/cudnn-11.2-windows-x64-v8.1.1.33.zip>
2. Download CUDA 11.2: <https://developer.nvidia.com/cuda-11.2.0-download-archive>
3. Restart your computer.
4. Ensure that C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2 exists.
5. Extract CuDNN files from ZIP, and copy bin, include and lib directories to the path mentioned in the previous step.
6. Ensure that your system environment variables has the following keys:
   1. CUDA\_PATH - C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2
   2. CUDA\_PATH\_V11\_2 - C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2
   3. In Path enum variable:
      1. C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\bin
      2. C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\libnvvp

# 4. Matplotlib on Colab – Headless Mode

Since our system uses Object-Oriented Classes and must run end-to-end in headless execution mode, we patch the backend:  
  
import matplotlib  
matplotlib.use("Agg")  
  
This enables WandB or file-based plot outputs instead of GUI rendering.

# 5. Data Files and JSON Structures

The following table outlines the key data files used throughout the authorship verification pipeline, their purpose, and the exact data structures they store. These files are essential for persistence, model communication, and cross-step consistency.

| **File Name** | **Purpose** | **Data Structure** |
| --- | --- | --- |
| **dataset\_impostors.json** | List of impostor authors which are listed in pairs.json, and their text corpora | [{"author": str, "texts": [str]}, ...] |
| **dataset\_all\_impostors.json** | Identical to the previous json, but is not limited to impostors listed in pairs.json only. | [{"author": str, "texts": [str]}, ...] |
| **included\_text\_names.json** | List of Shakespearean documents included in the test corpus. | [str, ...] |
| **pairs.json** | Contains valid author pairs for which models are trained; also manages iteration state. | {"pairs": [[str, str], ...], "models\_to\_skip": [...], "last\_iteration\_training": int, ...} |
| **distance\_matrix.json** | Distance matrix of DTW distances between signal pairs. | List[List[float]] |
| **clustering\_results\_all.json** | Clustering results and medoids after k-medoids clustering. | {"n\_clusters": int, "model\_features\_used": [...], "cluster\_assignments": {str: int}, "medoid\_texts": [...]} |
| **anomaly\_scores.json** | Anomaly scores per text using Isolation Forest. | {str: float} |
| **anomaly\_report.txt** | Plain-text readable anomaly report used to visualize and summarize detected outliers. | Plain text |
| **model\_name-signals.json** | Signals extracted from each text using Siamese networks trained on Agatha Christie vs Francis Bacon. | {str: List[float]} |
| **all\_models\_isolation\_forest\_score.json** | Aggregate scores across all impostor pairs for final clustering input. | {str: List[float]} |

# 6. Preprocessing Pipeline and Tokenization

Preprocessing is a foundational step in the procedures. It is responsible for transforming raw literary texts into tokenized, chunked, JSON-encoded formats that are later consumed by the Siamese model and surrounding procedures (e.g., DTW, classification, anomaly detection).

## Overview

This system employs two types of preprocessing strategies, depending on the stage of the pipeline and the purpose of the data:

## General Preprocessor

Defined in `src/preprocess.py` and instantiated in `Procedure.\_\_init\_\_()` as:  
  
 self.general\_preprocessor = Preprocessor(config=config)  
  
 This general preprocessor:  
 - Uses the vanilla BERT tokenizer (from HuggingFace Transformers)  
 - Does not require a fine-tuned model checkpoint  
 - Is used mainly for utility steps and shared preprocessing that doesn't rely on a specific impostor  
  
 Usage Examples:  
 - Preprocessing Shakespeare texts  
 - Preparing datasets for visualization  
 - Input texts for `text\_to\_classify.json`

## Fine-Tuned Tokenizer Per Impostor

When training or performing inference for a specific impostor pair, the system uses a tokenizer loaded from that impostor’s fine-tuned BERT. This is handled in `Procedure.\_\_load\_tokenizer\_and\_model(impostor\_name)`:  
  
 tokenizer = BertTokenizer.from\_pretrained(local\_model\_path)  
  
 This ensures alignment between the tokenizer's vocabulary and the domain-specific fine-tuned model.  
  
 Usage Examples:  
 - Siamese model training per impostor pair  
 - Generating signals for DTW  
 - Classifying unknown texts using a specific pair’s model

## How Preprocessing Works – Technical Flow

Source File: `src/preprocess.py`  
  
 Implements the `Preprocessor` class with methods like:  
 - `clean\_text(text)`: Normalize text  
 - `tokenize(text)`: Tokenizes using transformers.BertTokenizer  
 - `chunk\_text(text, chunk\_size)`: Creates overlapping BERT chunks  
 - `preprocess\_directory()`: Applies cleaning, chunking, and outputs JSON  
  
 Each chunk is padded, truncated to `max\_sequence\_length`, and stored in a list for usage.

## Data Flow Through Preprocessing

| Step | Method | Output |
| --- | --- | --- |
| Clean raw text | Preprocessor.clean\_text() | Normalized string |
| Tokenize | tokenizer.tokenize() | List of WordPiece tokens |
| Convert to IDs | tokenizer.convert\_tokens\_to\_ids() | List of token IDs |
| Chunk | Preprocessor.chunk\_text() | Chunked token segments |
| Save | to JSON | Used in dataset JSONs |

## Use Cases by Scenario

Scenario A: Training  
 - Uses fine-tuned tokenizer per impostor  
 - Prepares data using chunk\_size and token IDs for training Siamese model  
  
 Scenario B: Classification  
 - Uses general or fine-tuned tokenizer  
 - Converts arbitrary input into valid Siamese input format  
  
 Scenario C: DTW  
 - Generates signals from encoded chunk sequences  
 - Uses the same tokenizer used during model training  
  
 Scenario D: CLI Dataset Generation  
 - Converts all `.txt` into JSON via general tokenizer  
 - Saves output as `dataset\_shakespeare\_collection.json`, etc.

## Summary

- Modular, tokenizer-aware, and configurable per impostor pair  
 - Ensures tokenizer-model alignment  
 - Converts raw text into chunked BERT input for training and analysis  
 - Reused in training, DTW, classification, and dataset preparation

# 7. Inference & Training Overview

Each impostor pair spawns a Siamese Network, trained using Binary Cross Entropy.

An instance of SiameseBertModel from [model.py](http://model.py) is initialized for each impostor pair, where the impostor names are provided, and if the model is initialized for inference and not training, then use\_pretrained\_weights=True is supplied as an argument.  
This ensures that when get\_encoder\_classifier() is called, the weights of the latest model saved as an artifact in W&B are loaded.

During training, we use the instance to call build\_siamese\_model(bert\_model\_1, bert\_model\_2) - providing the fine-tuned berts of each impostor. Impostor 1 uses branch 1, therefore bert\_model\_1 is impostor 1’s fine-tuned BERT model. The same is said for impostor 2.

# 7. Hugging Face Integration

We host fine-tuned BERT models per impostor under the Hugging Face model hub:  
🔗 https://huggingface.co/ElyMK1/bert-shakespeare-english-mlm  
  
Each impostor's fine-tuned model is uploaded under a subfolder and referenced in:  
  
bert:  
 repository: "ElyMK1/bert-shakespeare-english-mlm"  
  
  
Models are downloaded at runtime using:  
- `huggingface\_hub.snapshot\_download`  
- Cached locally under `saved\_trained\_models/berts/{author}/`

# 8. WandB Logging & Artifacts

We use Weights & Biases (WandB) for:  
- Tracking loss, accuracy curves, and training metrics  
- Saving t-SNE plots of signal embeddings  
- Persisting trained models and clustering results  
- Saving all Matplotlib plots from headless environments  
  
WandB configuration:  
wandb:  
 enabled: True  
 project: "siamese-authorship-verification"  
 artifact\_name: "siamese-authorship-verification-branches"  
 api\_key: <<your api key>>  
  
Artifacts are saved under:  
<https://wandb.ai/authorship-verification-siamese-network/siamese-authorship-verification/artifacts/>

# 9. How to Test & Maintain

- Run `main.py` and follow the interactive menu  
- Use WandB dashboard for live monitoring  
- Add new impostor `.txt` files to `data/raw/impostors/`  
- Update or reconfigure model hyperparameters via `config.yaml`

- Choose specific impostor pairs by editing pairs.json; ensure that dataset\_impostors.json contains all the authors mentioned in pairs.json  
- Replace Isolation Forest or Clustering method easily via `src/` modules