Capstone Project Phase B

Analysis of Russian Texts Using Deep Impostor Method

25-2-R-12

By:
Daniel Feldman 208134700
Arthur Cherniy 206466500

Supervisors: Dr. Renata Avros, Prof. Ze'ev Volkovich

Link to GIT folder.

1. Problem and Background

The authorship of And Quiet Flows the Don, attributed to Mikhail Sholokhov, has been debated since its publication (1928–1940). Although the novel earned him the 1965 Nobel Prize in Literature, critics questioned whether its literary depth and historical detail were consistent with his young age and limited education. Early suspicions suggested he may have used manuscripts by Fyodor Kryukov, a Cossack writer whose background aligned with the novel's themes.

An official investigation in 1929 found no evidence of plagiarism and concluded that the novel's style matched Sholokhov's earlier work, Tales from the Don. Despite this, doubts persisted, often influenced by political arguments, including claims by Aleksandr Solzhenitsyn in the 1960s. Quantitative stylistic studies, notably by Kjetsaa (1984) and later Hjort (2007), found strong similarities between the novel and Sholokhov's confirmed writings and clear differences from Kryukov's works. Handwritten manuscripts discovered in the 1980s further documented Sholokhov's writing process.

Alternative theories have continued, including proposals of other possible authors. However, most literary, statistical, and historical evidence supports Sholokhov's authorship, while remaining skepticism tends to rely on weaker empirical support and political interpretation rather than textual analysis.

1.1. Word Embedding

Word embeddings are a natural language processing technique that represent words as dense numerical vectors, where semantically similar words are positioned closer together in a high-dimensional space. They are learned from large text corpora using machine learning models that analyze word context, allowing both semantic and syntactic relationships to be captured.

A common method for creating word embeddings is Word2Vec, which uses shallow neural networks through either the Continuous Bag of Words (CBOW) or Skip-Gram models to learn word representations based on surrounding context. These embeddings are widely used in NLP tasks such as text classification, translation, search, and question answering, as they enable models to understand language more effectively than traditional approaches.

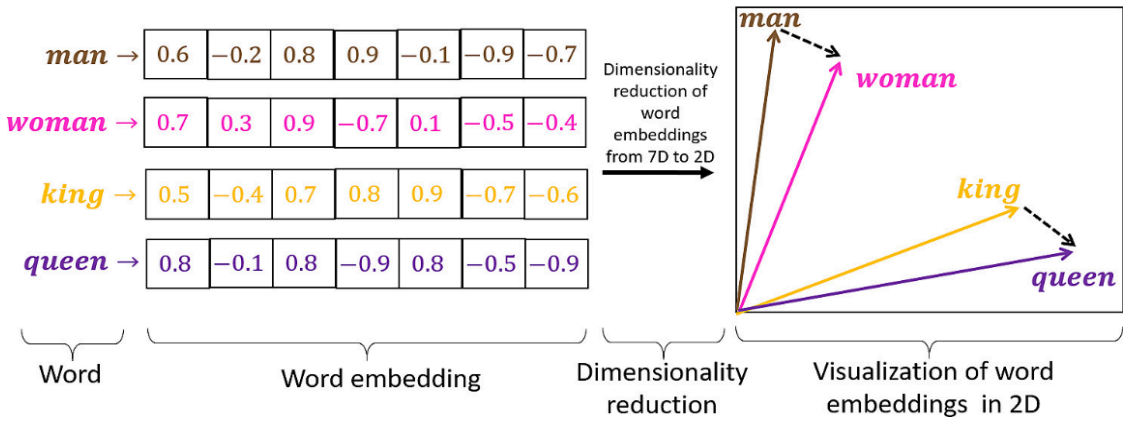


Figure 1. Visualization of word embedding in 2d space. Created by Palaniyappan T. (Medium).

1.2. Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a specialized type of artificial neural network designed to process data with a grid-like topology, such as images. CNNs are composed of several key layers: convolutional layers, pooling layers, and fully connected layers (O'Shea & Nash, 2015).

The convolutional layer is the fundamental building block of a CNN. It applies a set of learnable filters (kernels) that convolve across the input data, producing feature maps that capture local patterns such as edges or textures. Each filter is spatially small but extends through the full depth of the input, and as it slides across the input, it detects specific features at various spatial locations (O'Shea & Nash, 2015).

Pooling layers perform a downsampling operation along the spatial dimensions of the feature maps, reducing the dimensionality and the number of parameters in the network. This is typically achieved by applying an aggregation function, such as max pooling, over local regions of the input. Pooling helps to make the representations approximately invariant to small translations of the input (O'Shea & Nash, 2015).

After several convolutional and pooling layers, the high-level features extracted are fed into one or more fully connected layers. In these layers, each neuron is connected to every neuron in the previous layer. The final fully connected layer typically uses a softmax activation function to produce class probabilities for classification tasks (O'Shea & Nash, 2015).

By stacking these layers, CNNs can learn hierarchical feature representations, from simple edges in early layers to complex patterns in deeper layers, enabling effective analysis of visual and other grid-structured data (O'Shea & Nash, 2015).

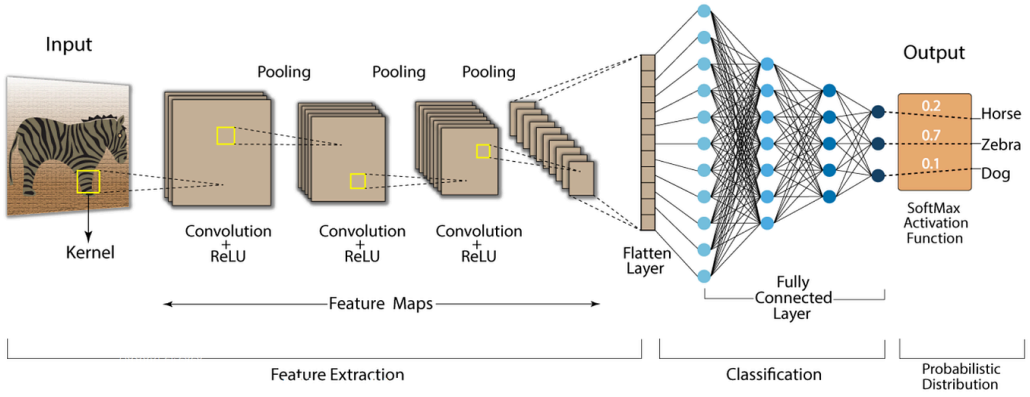


Figure 2. Illustration of a CNN. Created by Waqas, M. T. (Medium)

1.3. Long Short-Term Memory (LSTM)

A Long Short-Term Memory (LSTM) network is a type of Recurrent Neural Network (RNN) specifically designed to overcome the vanishing and exploding gradient problems that can occur when learning long-term dependencies in sequence data (Hochreiter & Schmidhuber, 1997). LSTMs introduce a memory cell and three gating mechanisms: input, forget, and output gates, that regulate the flow of information into, out of, and within the cell. This architecture enables the network to maintain and update information over extended sequences, making LSTMs particularly effective for tasks that require memory of past inputs.

The input gate controls which information is added to the memory cell, the forget gate determines which information is removed, and the output gate governs what information is output from the cell. By selectively retaining or discarding information, LSTMs can learn long-term dependencies in data.

LSTMs have demonstrated superior performance over standard RNNs in a variety of sequence learning tasks, including language modeling, speech recognition, handwriting recognition, time series prediction, and protein secondary structure prediction (Hochreiter & Schmidhuber, 1997).

1.4. Bidirectional Encoder Representations (BERT)

BERT (Bidirectional Encoder Representations from Transformers), introduced by Devlin et al. (2018), is a Transformer-based language model that improved NLP by using

bidirectional context. Unlike earlier unidirectional models, BERT processes text using both left and right context through self-attention mechanisms, enabling more accurate representations of meaning.

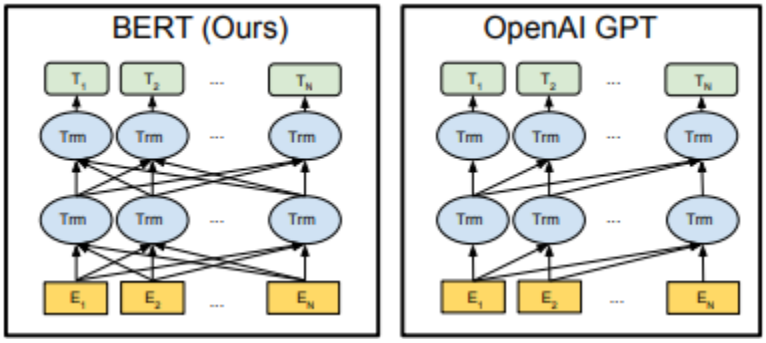


Figure 3. Illustration of BERT vs GPT architecture (bidirectional vs left-to-right). Devlin, J., et al. (2018).

BERT is pre-trained using two objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, 15% of tokens are selected for prediction, allowing the model to learn deep syntactic and semantic relationships from surrounding context. NSP trains the model to determine whether one sentence logically follows another, helping it capture inter-sentence relationships.

A key contribution of BERT is its contextualized embeddings. Unlike static embeddings such as Word2Vec or GloVe, BERT generates dynamic representations that depend on context. Text is tokenized using WordPiece, and each token is represented by the sum of token, segment, and positional embeddings before passing through multiple Transformer encoder layers. The final output provides token-level embeddings and a sentence-level embedding from the [CLS] token for classification tasks.

For Russian, specialized models such as ruBERT and ruRoBERTa have been developed. ruBERT, introduced by DeepPavlov and later extended by Zmitrovich et al. (2023), includes base and large versions trained on Russian corpora using MLM and NSP, making it well-suited for Russian-language tasks.

1.5. BERT Fine Tuning

Fine-tuning BERT involves taking a pre-trained BERT model and further training it on a smaller, task-specific dataset to adapt it for a particular application (Devlin et al., 2018). While BERT is initially pre-trained on large, general-domain corpora using self-supervised objectives, fine-tuning allows the model to specialize in tasks such as

sentiment analysis, question answering, or domain-specific language processing. During this process, the model's weights are updated to optimize performance for the target task while retaining the general language understanding acquired during pre-training. Fine-tuning is crucial because it enables BERT to learn the specific vocabulary and objectives of the downstream task, resulting in significantly improved accuracy and effectiveness for specialized applications (Devlin et al., 2018).

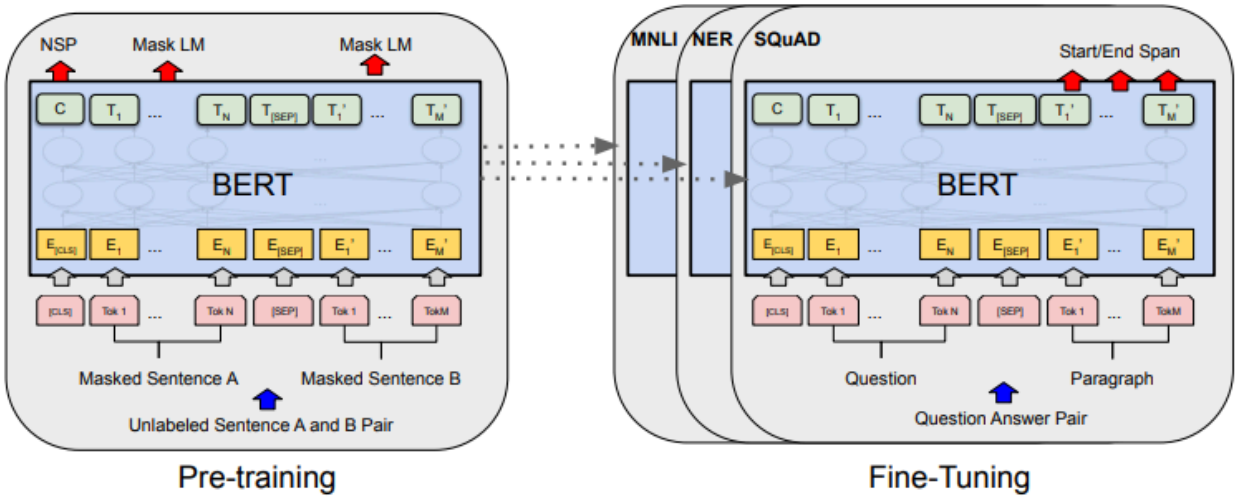


Figure 4. Illustration of BERT stages, pre-training and fine-tuning. Devlin, J., et al. (2018).

Fine-Tuning BERT for Authorship Attribution via Linguistic Style

This study fine-tunes a pre-trained BERT model for authorship attribution by leveraging its deep contextual embeddings to capture subtle differences in linguistic style. Since BERT is trained on large-scale language modeling tasks, its representations are well suited for identifying nuanced patterns in text that distinguish between different authors.

The approach follows methods commonly used in sentiment analysis, where a classification layer is added to the pre-trained model and the entire network is fine-tuned on labeled data. Authorship attribution is treated as a binary classification task, with texts from different author groups assigned opposite labels to emphasize stylistic contrasts. The objective is for the model to learn these differences and generalize them to unseen texts.

1.6. DTW Distance

Dynamic Time Warping (DTW) is a method for measuring similarity between two time series that may differ in length or speed. It aligns the sequences by computing pairwise distances between their points and finding the path that minimizes the total accumulated distance. This process allows one sequence to be stretched or compressed along the time axis, enabling effective comparison even when patterns are temporally misaligned.

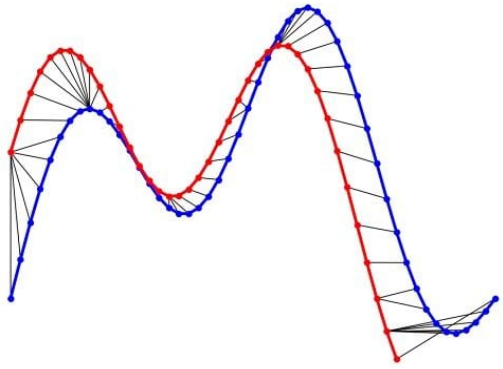


Figure 3. Illustration of DTW distance between two time series. Alizadeh E. (Ealizadeh).

1.7. Isolation Forest Algorithm

Isolation Forest is an unsupervised anomaly detection algorithm that identifies outliers by recursively partitioning data using randomly constructed trees (Liu, Ting, & Zhou, 2008). By randomly selecting features and split values, the algorithm isolates data points through successive partitions, where anomalous points are typically isolated with fewer splits. This property allows Isolation Forest to efficiently detect anomalies in high dimensional data without requiring labeled examples.

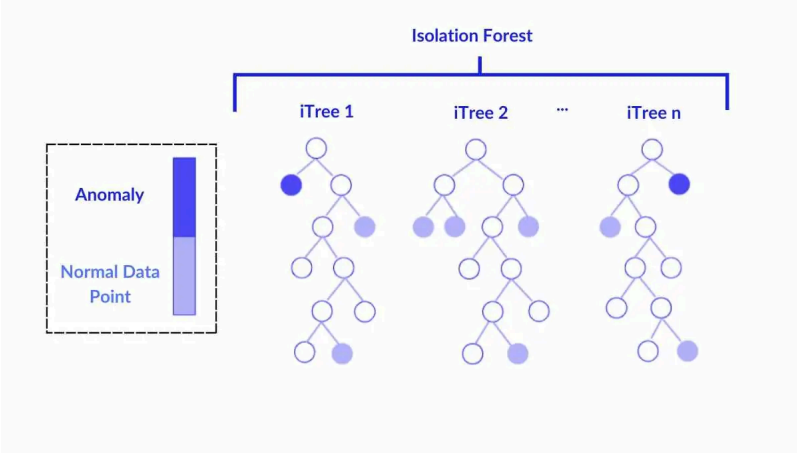


Figure 4. Illustration of Isolation Forest Algorithm. Created by Wrench. (Medium).

2. Solution Description

2.1 General Description

2.1.1 System Goals and Implementation

The goal of the system is to provide an end to end pipeline for authorship analysis of Sholokhov's texts using modern machine learning and anomaly detection techniques.

The system is composed of four core modules that work together to transform raw literary text into stylometric features and analytical results, ultimately providing insight into whether Sholokhov is the true author of the book collection attributed to him.

The Preprocessing Module:

The Preprocessing Module prepares the input books for modeling by cleaning and standardizing the text. This includes removing non-essential characters and normalizing whitespace to ensure a consistent text format across all inputs.

The Preprocessing Module was tested using unit tests to verify correct text cleaning behavior and error handling. The tests focus on ensuring that valid inputs are processed correctly and invalid inputs are handled safely.

preprocess_string(text):
This function was tested with a regular multi-line Russian text to verify that punctuation, extra whitespace, and non-essential elements are removed, resulting in a standardized output string. It was also tested with invalid inputs such as integers, lists, and dictionaries to ensure that a ValueError is raised.

read_with_best_encoding(path):
This function was tested using a valid temporary file encoded in UTF-8 to confirm that the text is read and decoded correctly. An additional test with a non-existent file path verified that the function raises a FileNotFoundError when the file cannot be found.

The Prediction Module:

The Prediction Module transforms the books into numerical signals by training a ConvBiLSTM model and fine tuning ruBERT to distinguish between impostor texts across impostor pair iterations. The trained models are then used to predict Sholokhov's book segments. Every 8 consecutive segment scores are averaged into a single chunk score, and these scores are concatenated to form a signal.

For testing, the prediction module, which consists of multiple classes and functions, was evaluated using unit tests to verify correct functionality and error handling.

ensure_directories():
This function was tested in scenarios where the required directories did not exist to confirm they were created successfully. It was also validated to ensure no errors occur when the directories already exist.

load_texts_from_dir(path):
This function was tested with a non-existent directory to verify that a FileNotFoundError is raised. It was also tested with mixed file types to ensure that only non-empty .txt files are loaded in sorted order.

TextSegmenter class:
The class was tested for successful initialization with valid input and correct assignment of configuration values. Invalid inputs were used to confirm that a ValueError is raised.

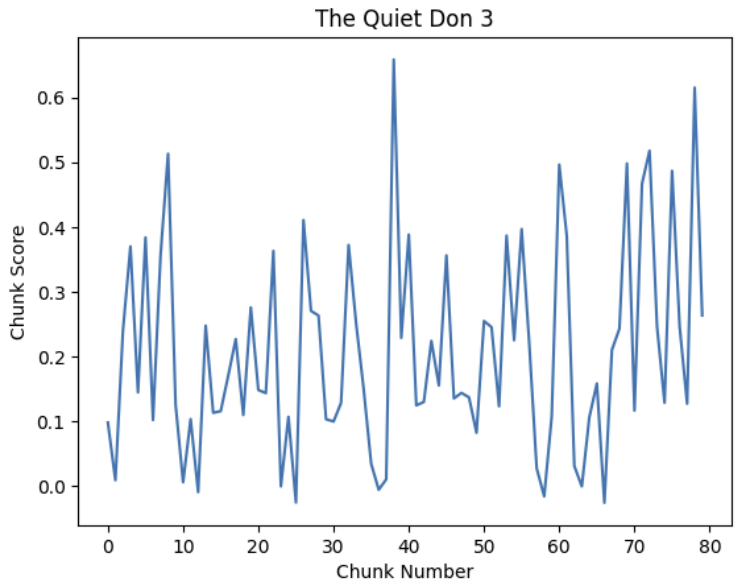


Figure 5. Illustration of a created signal.

The Clustering Module

The Clustering Module computes the DTW distance between all signals within each impostor pair, forming a distance matrix for each iteration.

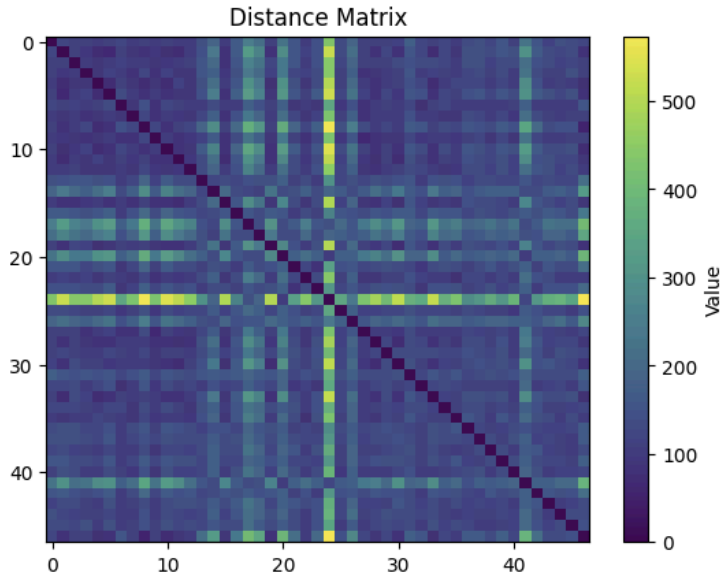


Figure 6. Illustration of a Dynamic Time Warping distance matrix.

Isolation Forest is then applied to each distance matrix to generate an anomaly vector, where each text receives an anomaly score. These anomaly vectors are aggregated into an anomaly matrix, which is clustered using the K-Medoids algorithm into 2 clusters based on randomly selected features. This process is repeated for 500 iterations, assigning each text a label per iteration: 1 for suspected fake and 2 for authentic. Finally using the mean score to produce the result. The final result is obtained by averaging the labels across all iterations, producing a score between 1 and 2, with 1.5 used as the decision threshold.

This module was tested using unit tests to validate DTW distance computation, distance matrix construction, and anomaly matrix generation. The tests cover both correct outputs for valid inputs and raising exceptions for invalid inputs.

calculate_dtw(self, signal1, signal2):
The DTW calculation was tested with two valid numeric signals to confirm that a finite, non negative distance value is returned. It was also tested with invalid inputs (e.g., non list values) to verify that the implementation raises a ValueError.

create_distance_matrix(signals, dtw_service):
This function was tested with a valid list of signals and a valid DtwService implementation to confirm that it returns an n × n symmetric matrix with zeros on the diagonal. Additional tests used invalid signals (e.g., wrong types or incompatible signal

structures) and an invalid dtw_service (not implementing DtwService) to ensure the function raises a ValueError rather than producing incorrect output.

build_anomaly_matrix(signal_dir, dtw_service):
This function was tested using a valid directory containing *_signals.json files to confirm that it correctly loads signals, builds per iteration distance matrices, computes anomaly scores, and returns (book_names_ordered, anomaly_matrix, times) with consistent dimensions. It was also tested with an invalid signal directory (missing or unreadable path) and an invalid dtw_service to verify that appropriate exceptions are raised and failures are handled predictably.

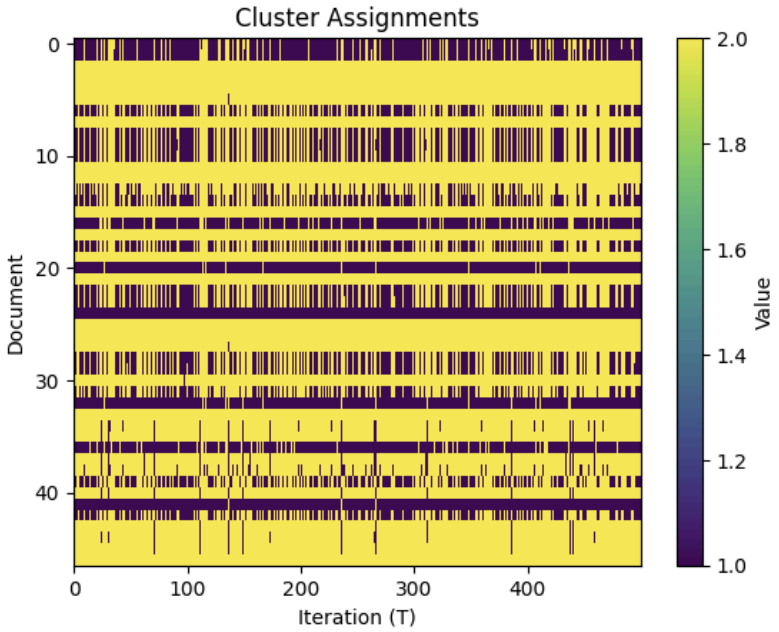


Figure 7. Illustration of a K-Medoids cluster assignments across 500 iterations.

The Scraper Module:
The Scraper Module downloads impostor texts from lib.ru and serves as an automation tool for the repetitive task of collecting Russian language texts and storing them in the project drive.

Starting from the base URL: “https://lib.ru/PROZA/”, the scraper downloads and parses the HTML content to extract links to author directories and their book collections. It processes a limited number of authors, skipping those that were already downloaded, and for each valid author it scans the directory for text files. Books are accessed directly via URLs of the form “https://lib.ru/PROZA/AUTHOR_NAME/BOOK_NAME.txt” and are downloaded one by one. During download, the HTML content is cleaned to remove non text elements, and the extracted text is saved as a UTF-8 txt file. Each author is

assigned a dedicated folder in the output directory, and the process concludes with a verification step that confirms successful downloads.

This module was tested using basic unit and integration tests to ensure correct behavior for valid inputs and graceful handling of invalid cases, without using external testing frameworks.

_sanitize_path_component(name)
Tested with valid filenames to ensure proper cleaning and formatting, and with invalid strings to verify that a safe default name is returned.

_get_soup(url, encoding)
Tested with a valid URL to confirm successful HTML parsing, and with an invalid URL to ensure the function fails gracefully and returns None.

_extract_text_content(soup)
Tested with HTML containing real text and noise to verify correct text cleaning, and with missing content to ensure the function safely returns None.

scrape_site()
Tested with an invalid root URL to confirm safe termination without side effects, and with zero authors configured to ensure no folders or files are created.

2.1.2 User Identification

The target users were identified as researchers and individuals interested in the authorship debate surrounding And Quiet Flows the Don, particularly scholars in computational linguistics, literary studies, digital humanities, and data science who seek quantitative methods to support textual analysis. It may also serve graduate students and independent researchers exploring stylometry or authorship attribution, providing them with an end to end pipeline for systematic text comparison, reproducible experimentation, and objective evaluation of stylistic similarity in addressing literary and historical questions.

2.2 Solution Description:

2.2.1 Description of Research and Development

The research was conducted using Google Colab as the experimental environment, leveraging PyTorch and GPU acceleration to efficiently train and evaluate the models. We first developed the core model that served as the foundation for the authorship authentication task. In the initial stage, our focus was on predicting text segments and generating stylometric signals, without applying clustering, in order to validate the model's ability to capture meaningful linguistic patterns.

To build a sufficiently large corpus of Russian literary texts, we developed a scraper service to systematically collect the required data. After preparing and organizing the dataset, we implemented the training procedures and conducted experiments using the impostor pairs framework. This process generated signals representing stylistic relationships between texts.

In a later stage, we introduced the clustering algorithm to analyze structural relationships within the generated signals. We also implemented visualization techniques, including one dimensional scatter plots, to clearly present and interpret the results.

The system was designed with flexible and easily adjustable parameters, allowing us to modify variables such as the number of impostor pairs, words per segment, and segments per chunk across different experiments.

Throughout the research process, results were regularly evaluated and presented to the supervisors, leading to new objectives for each development sprint. This iterative approach ultimately resulted in significant refinements, including a complete reconstruction of components such as the clustering module.

The Deep Impostor Method

The Deep Impostor Method proposed by Volkovich and Avros is a deep learning based approach to authorship analysis. It splits both the tested texts and impostor texts into equal word segments and trains a neural model to distinguish between different impostor styles. The trained model then assigns each segment of the tested texts to one of the impostors, turning each document into a numerical signal based on these assignments. These signals are compared using Dynamic Time Warping, and an Isolation Forest algorithm is used to detect stylistic outliers. Finally, clustering separates texts into likely authentic and suspected non-authentic groups by identifying deviations from the main writing style.

Algorithm

Signal Creation

The algorithm converts each of Sholokhov's works into a numerical signal by first dividing all texts into fixed length segments. For each randomly selected impostor pair, a ConvBiLSTM + BERT model is trained to distinguish between text segments written by the two impostor authors. The trained model is then used to score the segments of Sholokhov's texts, producing a sequence of values that reflect stylistic similarity to each impostor. Every eight consecutive segment scores are averaged into a single chunk score. These chunk scores are then concatenated in their original order, forming a continuous signal that represents the stylistic structure of the entire work for the given impostor pair.

Input:

- D: collection of test documents
- Z = {Z1, ..., Zm}: impostor collections
- S: supervising (separating) algorithm
- L: batch length
- k: number of batches per chunk

Output:

- Signal representations of the documents under consideration

Procedure:

1. Divide each document d in D and each impostor text z in Z into sequential batches of length L.
2. Train the supervising algorithm S on the batches of the impostor texts to obtain a separator S_ij that distinguishes between different impostor collections.
3. For each document d in D do:
    a. Initialize B(d) as an empty list.
    b. For each batch b of document d do:
        i. Compute the score of batch b using S_ij.
        ii. Append the score to B(d).
    c. Compute the signal Sig(d) for each chunk by averaging the scores of the k batches belonging to that chunk.
    d. Concatenate the chunk-level signals to form the final signal representation Sig(d).

Clustering Algorithm

This algorithm performs authorship verification by repeatedly comparing documents under randomly selected impostor conditions. In each iteration, a random subset of

features is selected and the documents are clustered into two groups based on these features. Final authorship scores are obtained by averaging the cluster labels across all iterations, yielding a stable verification measure that reflects consistent stylistic separation.

Input:

- A: an n x N matrix of document scores, where n is the number of tested documents and N is the number of performed iterations.
- T: number of iterations (steps).
- m: number of randomly chosen impostor pairs per iteration.

Output:

- Final authorship decision scores for the tested documents.

Procedure:

1. Repeat T times:
    a. Randomly select m features without replacement from the N available features.
    b. Cluster the n documents into two clusters based on the selected features.
    c. Align the resulting cluster labels to ensure consistent labeling across iterations.

2. For each tested document:
    a. Compute the average of its cluster labels across all T iterations.

2.2.2 Tools, Technologies and Supervisor Interaction

Python, PyTorch, DeepPavlov ruBert transformer model, Google Colab with A100 GPU, SciKit-Learn for Isolation Forest algorithm, DTW computation libraries, SciKit-Learn-Extra for K-Medoids, AI tools such as ChatGPT and Claude.

Regular meetings were held with our supervisors, during which we reviewed experiments, methodological challenges, model behavior, and result interpretation. Their guidance informed decisions regarding segmentation strategies, architecture choices, and the handling of overfitting during fine tuning.

2.2.3 Challenges, Outcomes and Conclusions

Challenges

1. Learning PyTorch and Transformer Fine-Tuning

A significant initial challenge involved acquiring practical proficiency in PyTorch and understanding how to fine tune large transformer models such as BERT. This required learning tensor operations, training procedures, and GPU aware implementation strategies.

2. Preprocessing Difficulties and Mojibake Correction

Many scraped texts contained mojibake and inconsistent Cyrillic encodings. To ensure reliable input data, we developed preprocessing methods for detecting corrupted encodings and normalizing all text into a consistent UTF-8 format.

3. GPU Memory Limitations and Performance Optimization

Fine tuning multiple BERT models led to frequent CUDA out of memory errors and prolonged runtimes. We addressed these issues through mixed precision training, gradient accumulation, improved memory management, and ultimately upgrading to an A100 GPU. Additional optimizations such as tokenization caching and reduced CPU to GPU data transfers, further improved performance.

4. Device Placement Constraints

BERT's strict requirement that all tensors reside on the same device caused recurrent errors during development. Overcoming this challenge required careful tracking of tensor placement and explicit device management throughout the pipeline.

5. Downloading the Corpus and Impostor Texts

Acquiring a corpus consisting of books from more than 50 authors, with approximately 10 books per author, proved to be a time consuming and repetitive task. This challenge was overcome by developing an automated web scraping service that downloaded the texts and stored them directly in the project drive.

6. Long Runtimes

Long runtimes were a major challenge, as Google Colab sessions could crash or stop unexpectedly despite offline execution support. To mitigate this, the system was divided into separate modules, where each module uses the output of the previous one. This reduced the runtime of each stage and lowered the risk of the session failing before completion.

Outcomes and Conclusions:
Despite the challenges, the project achieved its goals. We built a complete deep learning framework, successfully implemented the Deep Impostor method for Russian texts, and produced an independent, data driven perspective on the authorship debate. While our results only partially align with established scholarly conclusions, they contribute meaningful empirical insight and provide a foundation for future work in stylometric analysis.

Decision Making:
Throughout the project, decision making prioritized understanding and planning before implementation. Time was invested in evaluating approaches and selecting solutions that balanced performance, development effort, and expected value. A clear example of this approach is the development of the scraper service, which significantly reduced manual workload. In addition, knowledge acquired in prior courses, was leveraged where appropriate. Each decision was made with consideration for efficiency and time to value, allowing the project to be completed while balancing other academic responsibilities.

2.2.4 Lessons Learned

Looking back, completing a short PyTorch and transformer focused course at the beginning of the project would have made a noticeable difference. Having a stronger foundation before starting implementation would have saved time and made it easier to work with BERT, convolutional layers, and GPU based training.

Another key lesson was the importance of planning for memory usage and performance early in the development process. Considering GPU limitations, batching strategies, and caching mechanisms would have prevented many of the runtime and out of memory issues encountered later. Overall, the project emphasized the value of early preparation and careful resource planning when working on large scale deep learning tasks.

2.2.5 Meeting Project Metrics

The project goals focused on implementing the Deep Impostor method for Russian language texts, handling large scale literary data, and producing a reliable computational analysis of authorship.

To evaluate the system, clear performance metrics were defined:

- Achieving an average prediction accuracy above 80% across all impostor pairs.
- Completing book level prediction in under 10 seconds per impostor pair on average.
- Finishing a full impostor pair iteration in under 20 minutes.

These evaluation criteria were successfully met:

- The system reached an average prediction accuracy of 87%
- Book prediction takes 6.52 seconds per impostor pair on average
- Full impostor pair iteration completes in 12.34 minutes on average.

Beyond quantitative performance, the system also met its qualitative evaluation goals. The Deep Impostor method was fully implemented for Russian language material, and the pipeline demonstrated the ability to preprocess, segment, classify, cluster, and evaluate large volumes of text while producing interpretable stylistic signals. The results show partial alignment with conventional scholarly views on the authorship of the texts, indicating that “And Quiet Flows the Don” exhibits a distinct textual style relative to other works attributed to Sholokhov. Given the inherently interpretive nature of authorship attribution, such variation in conclusions is expected. Overall, the project met its defined evaluation metrics and provides a meaningful, data driven contribution to the ongoing authorship debate.

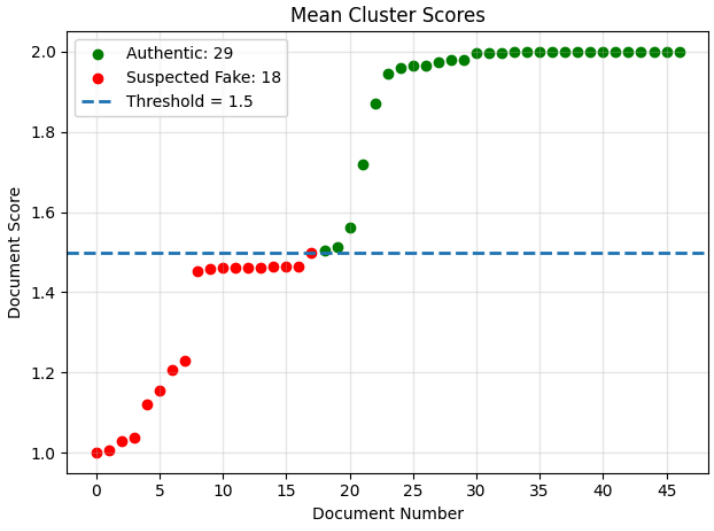


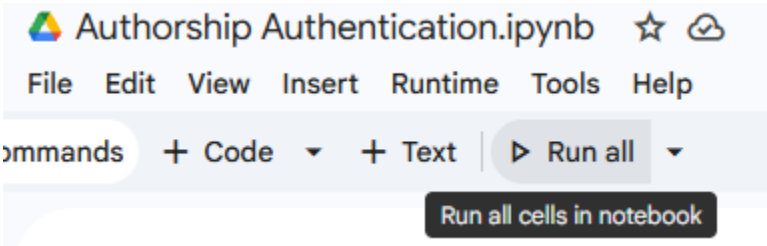
Figure 8. Illustration of final results.

User Guide

The full pipeline is long and may run for many hours, depending on available runtime hardware accelerators.

To run everything in order, you can use:

1) Colab menu: “Runtime” then “Run all”, note that this will start the scraper module in the end.

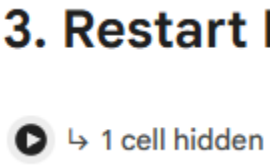


2) Run cells step-by-step, top to bottom, making sure each one finishes before moving on.

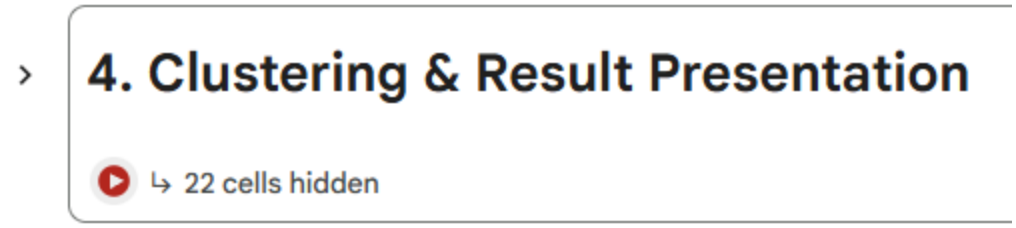
1. Run the Preprocessing Module: Preprocesses all texts, preparing the input for the experiment.

2. Run the Prediction Flow: The heaviest part of the system, turns the preprocessed texts into signals with the ML models.

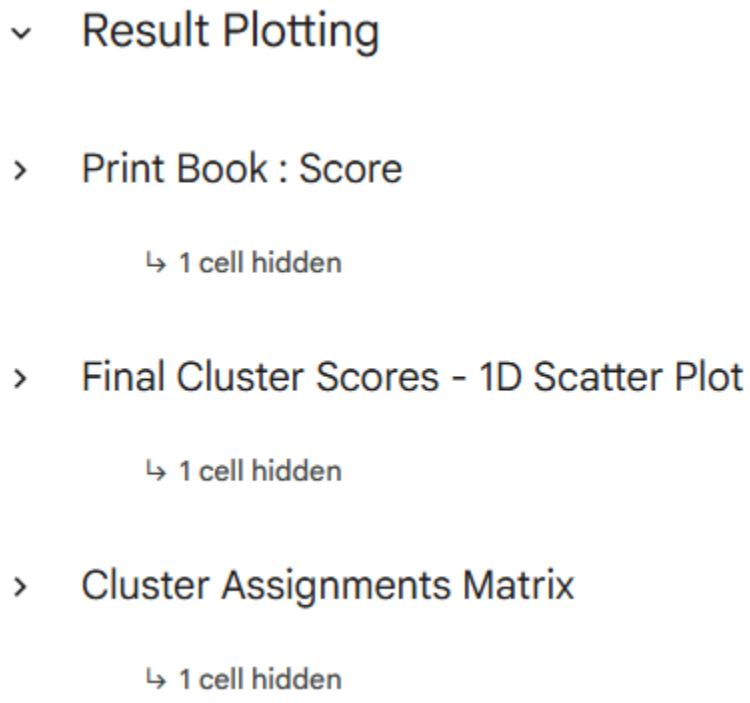
3. Restart Runtime: Runtime must be restarted to uninstall pre-installed libraries.



4. Clustering & Result Presentation: Loads the signals and clusters them.



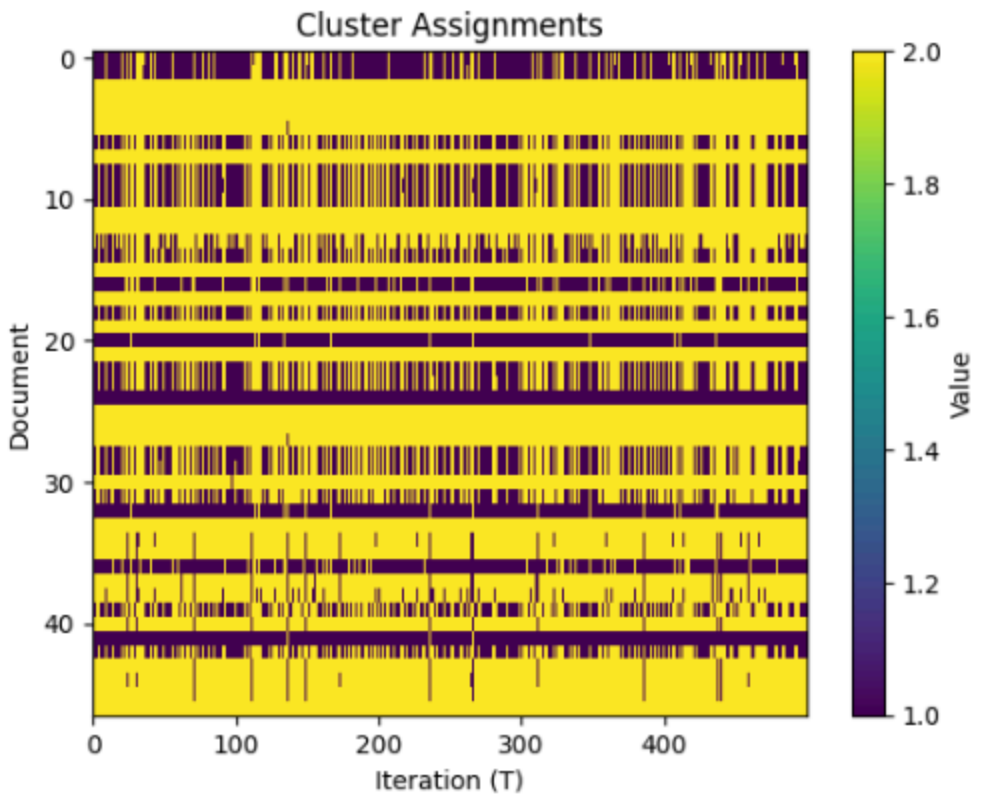
Results can be seen in the nested cells under “Result Plotting”:



Cluster Assignments Matrix

```
path = os.path.join(OUTPUT_DIR, "prediction_label_matrix.npy")
anomaly_matrix = np.load(path)
print(anomaly_matrix.shape)
print(anomaly_matrix.dtype)
plt.figure()
plt.imshow(anomaly_matrix, aspect="auto", interpolation="nearest")
plt.colorbar(label="Value")
plt.xlabel("Iteration (T)")
plt.ylabel("Document")
plt.title("Cluster Assignments")
plt.show()
```

```
(47, 500)
int64
```



Important Runtime Note

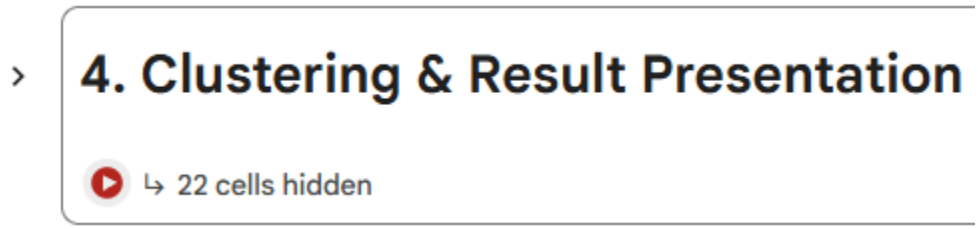
The session is expected to crash deliberately after running Cell 3.

This behavior is intentional and is used to:

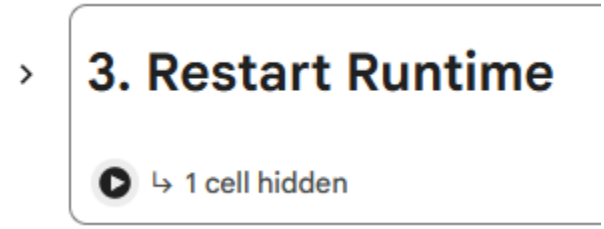
- Remove preinstalled Google Colab dependencies,
- Reinstall specific library versions required for clustering.

After the crash:

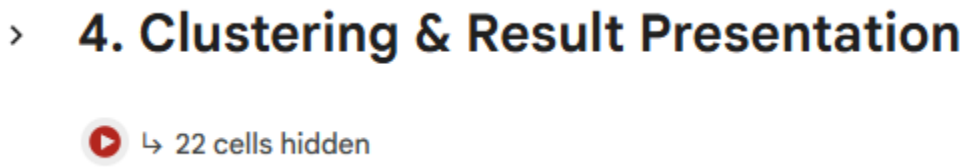
1. Run Cell 4.



2. Run Cell 3 again.



3. Run Cell 4 once more.



Maintenance Guide

Overview

This guide explains how to install the authorship verification pipeline using Google Colab with data stored in Google Drive. It covers environment setup, path configuration, and execution of the main experiment.

The pipeline is computationally intensive and may run for several hours. Using Google Colab Pro is recommended for faster execution, although the free version can still be used with reduced performance.

Installation

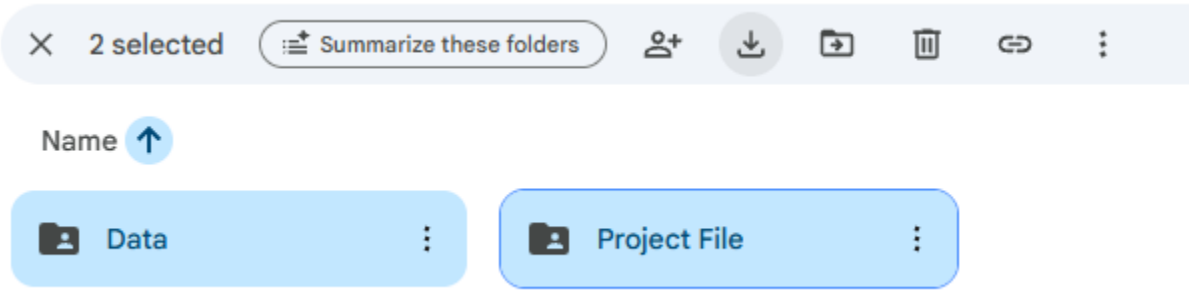
1. Prerequisites

Before you start, make sure you have:

- A Google account.
- Access to the project files at: https://drive.google.com/drive/folders/1ChFMvmXkx6SZ6YKq8mkSJEJXnncRJ3JQ?usp=drive_link
- Basic familiarity with:
  - Opening notebooks in Google Colab.
  - Navigating folders in Google Drive.

2. Getting the project into your Google drive

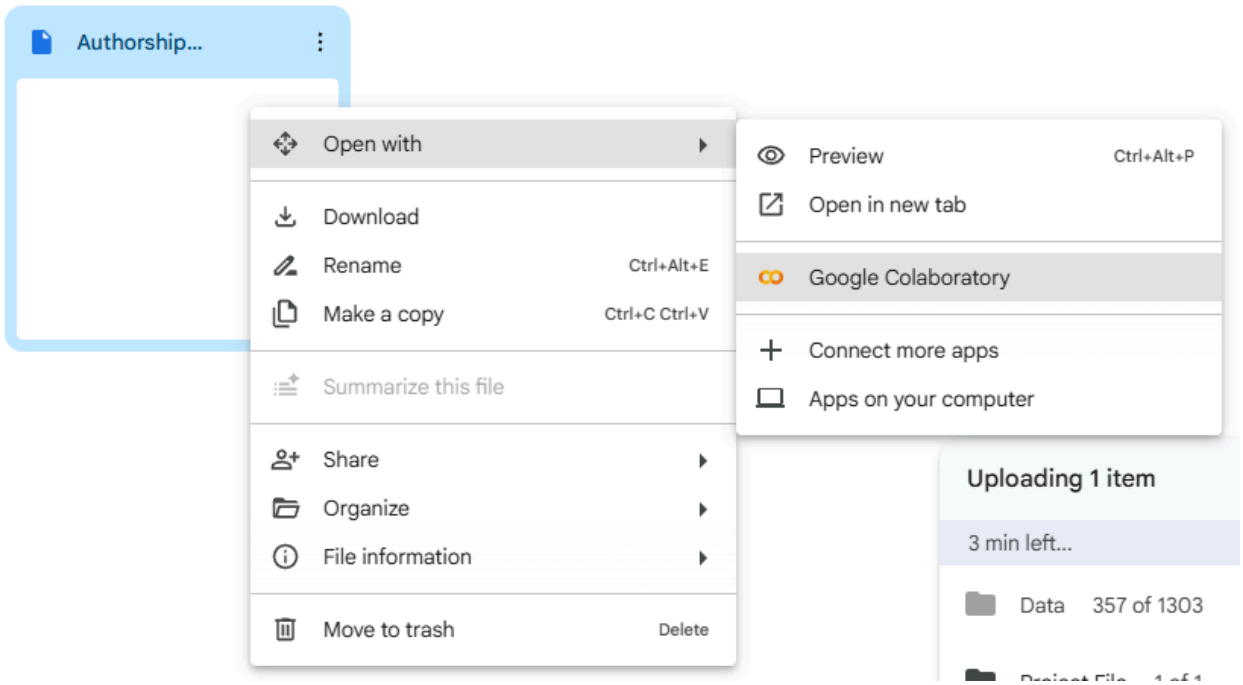
1. Open the Google Drive project link in your browser.
2. Download the project files to your computer.



3. In Google Drive, create a folder and name it "sholokhov_project".
4. Upload the downloaded files into your "sholokhov_project" folder in Google Drive.

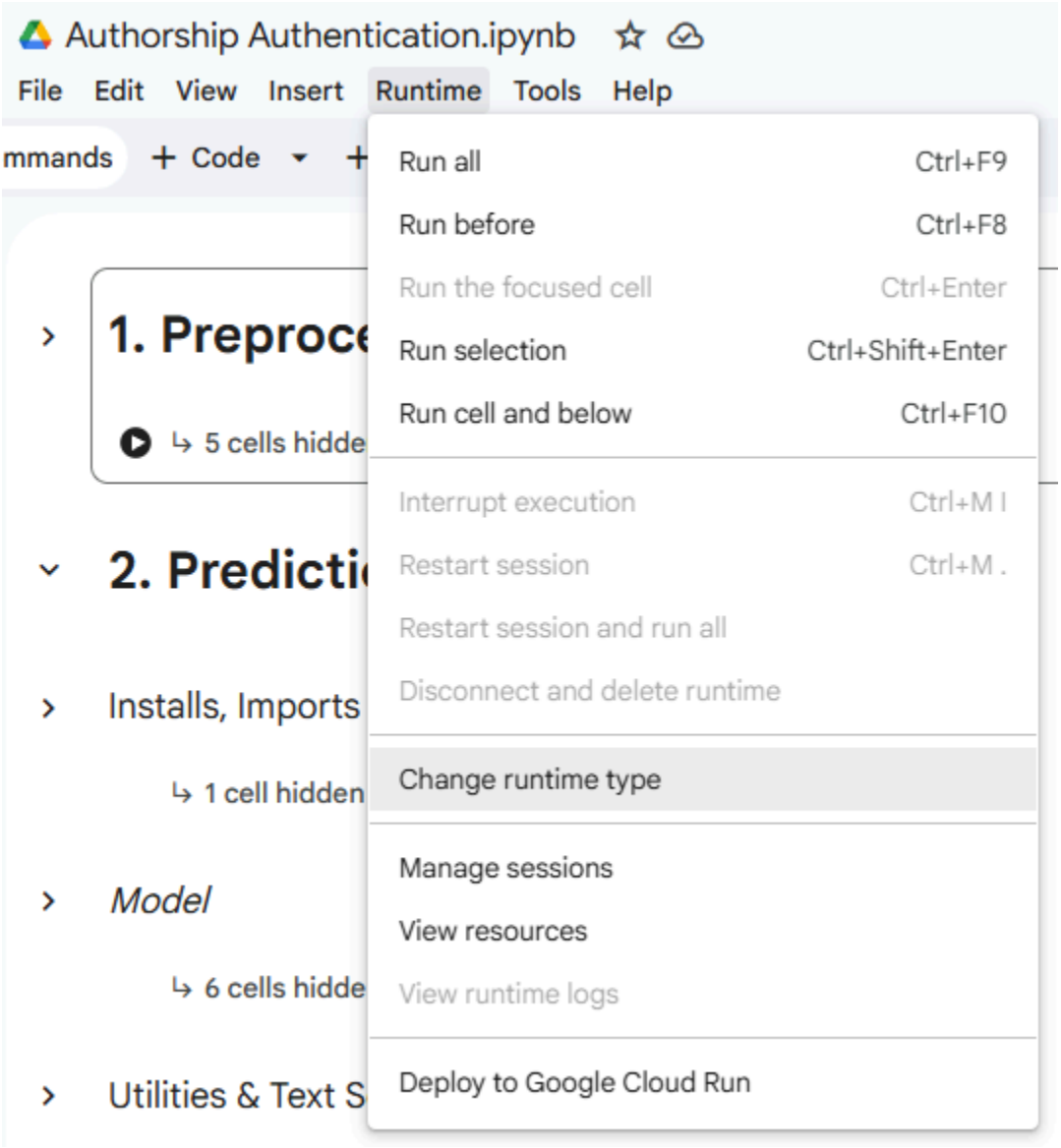
3. Opening the project Google Drive

1. In Google Drive, locate the project notebook file at “/Project File/Authorship Authentication.ipynb”.
2. Right-click the notebook.
3. Select “Open with” then “Google Colaboratory”.

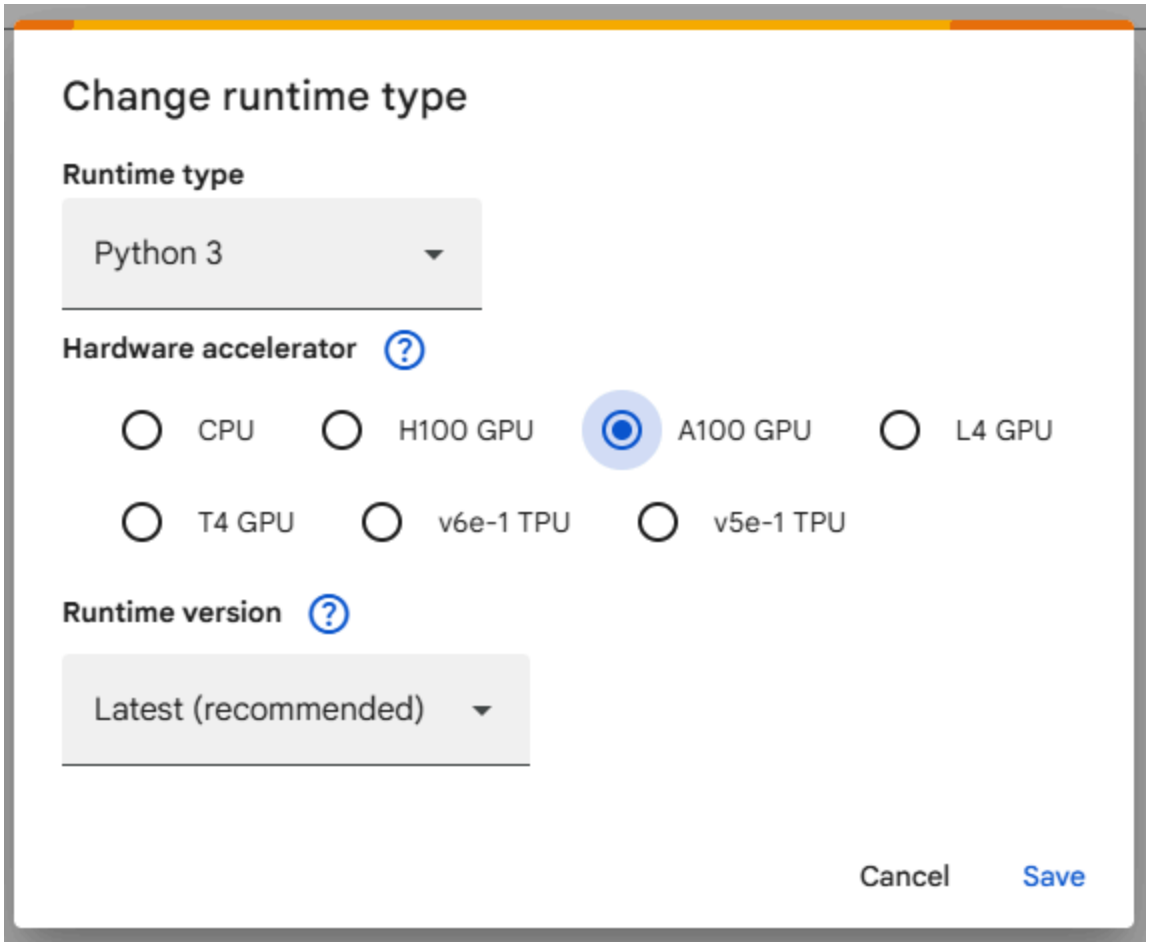


4. Setting up the runtime hardware

1. In Colab, open the “Runtime” menu.
2. Click “Change runtime type”.



3. Under “Hardware accelerator”, select
    a. “A100 GPU” - strongly recommended.
    b. “GPU” acceptable fallback if A100 is not available, or
    c. “TPU” (for v5e-1 TPU if available).



4. Click “Save”.

Note: The script uses PyTorch and transformers. GPU works reliably; TPUs may require extra setup. If you see TPU-related errors, switch to GPU.

5. Configuring the paths

The script expects your data and results to be under `/content/drive/MyDrive/...`. You need to adjust the paths to match your own folder structure.

Key path variables (look for them near the top of the script):

1. Preprocessing section
    a. `BOOK_COLLECTIONS_DIR = '/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Books'`
    b. `PROCESSED_SAVE_DIR = '/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Processed'`

```
BOOK_COLLECTIONS_DIR = '/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Books'
PROCESSED_SAVE_DIR = '/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Processed'
```

2. Prediction Flow section:
    a. `DATA_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Processed"`
    b. `SHOLOKHOV_PATH = os.path.join(DATA_DIR, "SHOLOKHOV")`
    c. `OUTPUT_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Signals"`

```
DATA_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Processed"
SHOLOKHOV_PATH = os.path.join(DATA_DIR, "SHOLOKHOV")
OUTPUT_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Signals"
OUTPUT_DIR = os.path.join(OUTPUT_DIR, EXPERIMENT_DATE_AND_TIME)
```

3. Clustering & Result Presentation:
    a. EXPERIMENT_DATE_AND_TIME = "3 Conv Layer - Impostor Hashing" - This must match the experiment date and time produced by the Prediction Flow module, failure to match this path will cause the clustering to work on different signals than the ones intended
    b. OUTPUT_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Signals"

```
EXPERIMENT_DATE_AND_TIME = "3 Conv Layer - Impostor Hashing"
OUTPUT_DIR = "/content/drive/MyDrive/Studies/Semester 8/Capstone Project - Phase 2/Data/Signals"
```

Important Note:
EXPERIMENT_DATE_AND_TIME must exactly match the experiment identifier produced by the Prediction Flow module. If it does not match, clustering will be applied to the wrong signal set.

Adapting Paths to Your Drive:
If you followed the previous steps, your folder path should be:
`/content/drive/MyDrive/sholokhov_project`.

otherwise:

1. In Google Drive, navigate to the folder where your `Data` directory lives.
2. In Colab's file browser (left sidebar), right-click a folder and choose "Copy path" to see its exact path.
3. Replace the example base path in the variables above with your own path, keeping the `/content/drive/MyDrive/...` prefix.

Software Dependencies

The system relies on external Python libraries for text preprocessing, deep learning, embedding generation, distance computation, clustering, and anomaly detection.

Main dependencies:

- NLTK – tokenization and Russian stopword removal
- chardet – automatic text encoding detection
- transformers – BERT tokenizer and model (DeepPavlov/rubert-base-cased)
- torch (PyTorch) – ConvBiLSTM model training and GPU execution
- fastdtw – approximate Dynamic Time Warping computation
- scipy – Euclidean distance function for DTW
- scikit-learn – evaluation metrics and Isolation Forest
- scikit-learn-extra – KMedoids clustering

Install all required dependencies using:

```
!pip install torch transformers nltk chardet fastdtw scipy scikit-learn scikit-learn-extra
```

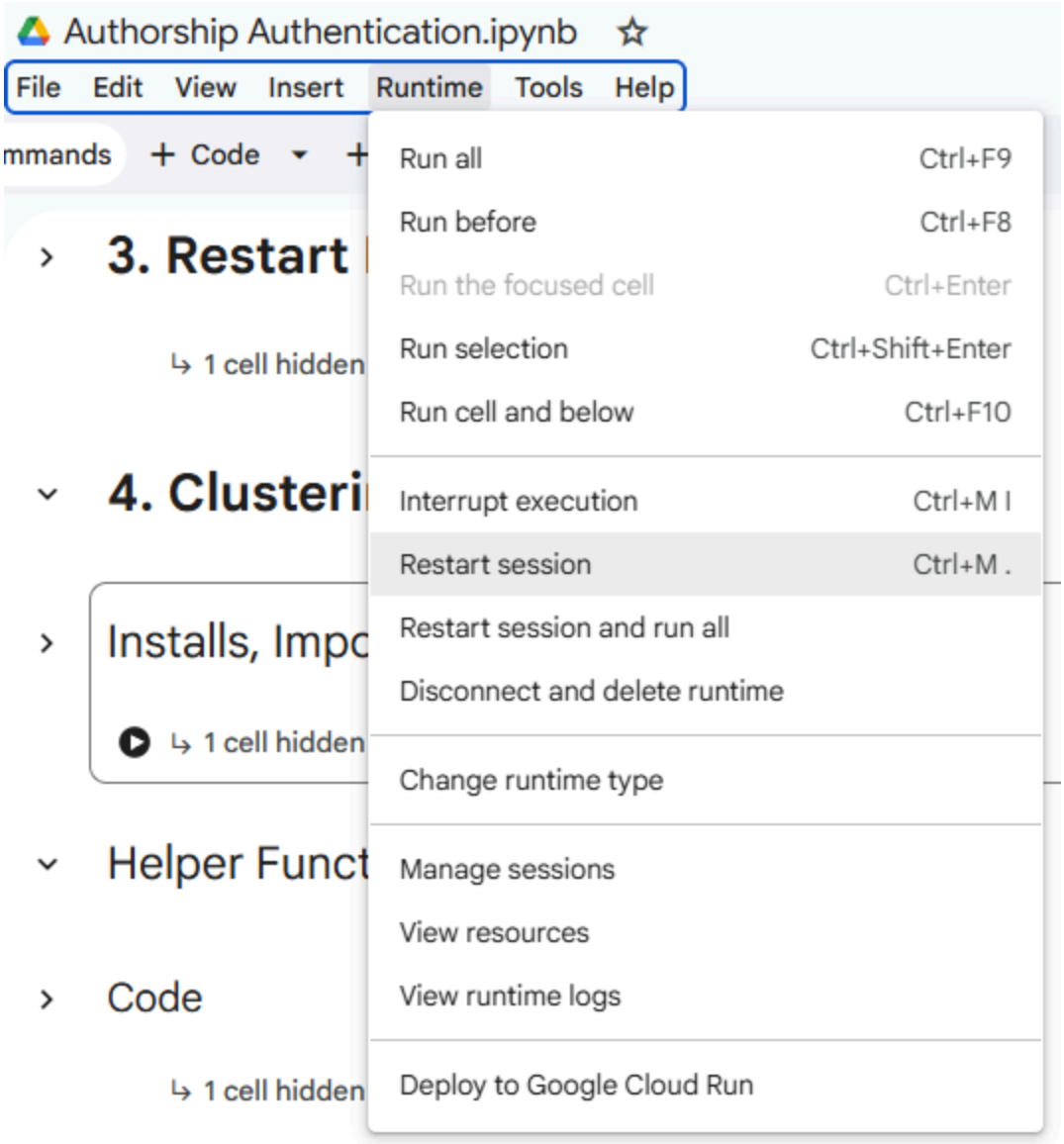
Updating the System

Over time, external libraries such as PyTorch, transformers, or scikit-learn may release new versions that introduce performance improvements or API changes. When updating dependencies, follow these steps:

1. Update packages explicitly using pip, for example:

```
!pip install --upgrade torch transformers scikit-learn scikit-learn-extra fastdtw
```

2. Restart the Colab runtime after upgrading to ensure all changes take effect.



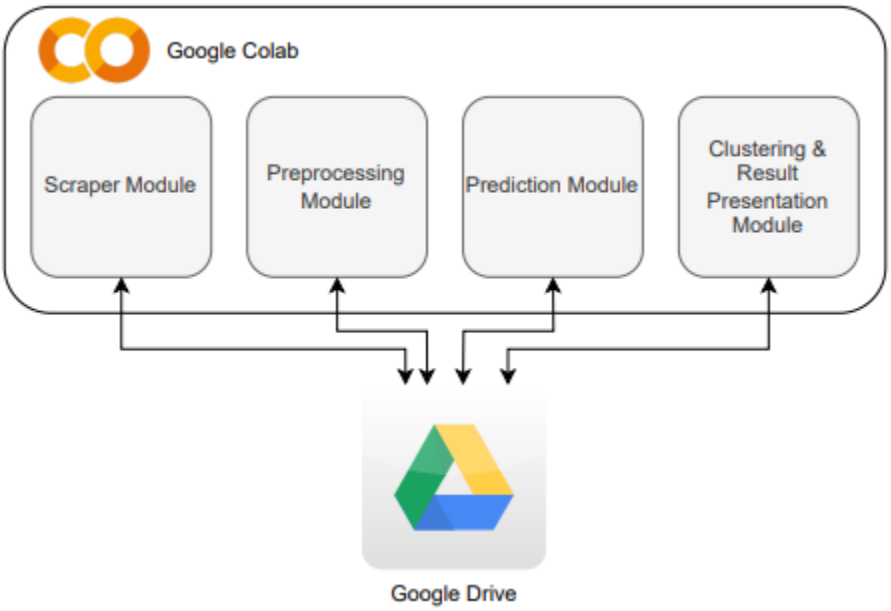
3. Re-run the notebook from the beginning and verify that:
   - The BERT model loads correctly.
   - The ConvBiLSTM training runs without shape or device errors.
   - Signal generation produces JSON files as expected.
   - Clustering and anomaly detection complete without exceptions.

4. If a library update causes breaking changes, reinstall a previously working version by specifying it explicitly, for example:

```
!pip install transformers==4.38.0
```

It is recommended to document the working library versions used for each major experiment to ensure reproducibility.

Architecture



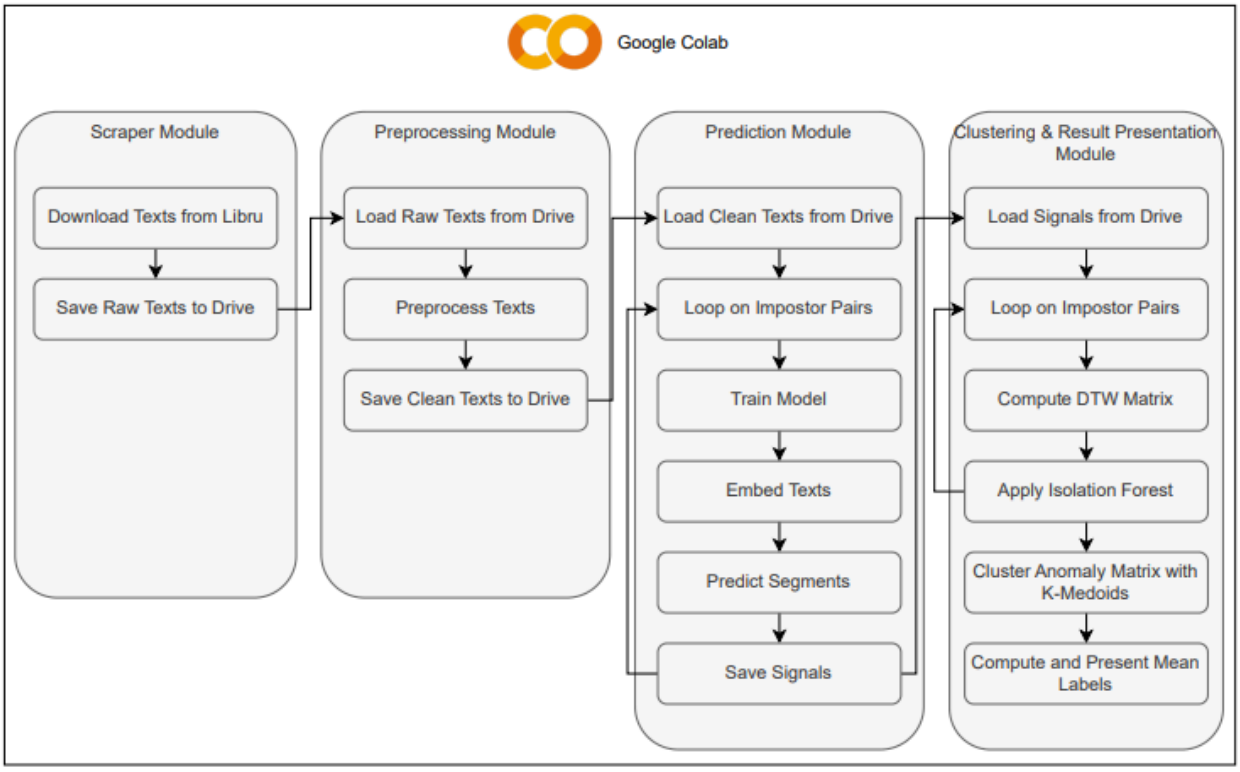
Scraper Module: The Scraper Module automatically collects impostor texts from lib.ru and stores them in a structured format in Google Drive. It parses author directories, downloads valid .txt files.

Preprocessing Module: The Preprocessing Module loads raw texts from Google Drive, cleans and standardizes them to ensure consistent input for modeling, and saves the processed outputs back to the Drive.

Prediction Module: The Prediction Module loads the cleaned texts from Google Drive and converts them into numerical signals using a ConvBiLSTM model and fine-tuned ruBERT. After generating segment-level and aggregated chunk scores, it saves the final signal representations back to the Drive.

Clustering and Result Presentation Module: The Clustering Module loads the signals from the Drive, computes DTW distance matrices from the generated signals and applies anomaly detection and K-Medoids clustering across multiple iterations. It averages the resulting labels to produce a final authenticity score and applies a threshold to determine whether a text is authentic or suspected fake.

Procedure Flow



The workflow begins with the optional scraping of impostor texts, which are downloaded and stored in Google Drive. The preprocessing module then loads the raw texts from the Drive, cleans and standardizes them, and saves the processed versions back to storage. Next, the prediction module loads the cleaned texts, trains the model across impostor pairs, generates segment level predictions, and saves a numerical signal for each book. Finally, the clustering module loads these signals, computes DTW distances and anomaly scores, performs clustering across multiple iterations, and produces the final authenticity result.