COMPARATIVE ANALYSIS OF DEEP LEARNING FRAMEWORKS FOR AI-GENERATED FAKE TEXT DETECTION [*]

Sai Vamsi Krishna Yedlapati
M.S. in Computer Science
Arizona State University
Tempe
syedlap1@asu.edu

Rohitha Somuri
M.S. in Computer Science
Arizona State University
Tempe
rsomuri1@asu.edu

Dyuti Mengji
M.S. in Computer Science
Arizona State University
Tempe
dmengji1@asu.edu

Pranay Reddy Palle
M.S. in Computer Science
Arizona State University
Tempe
ppalle3@asu.edu

ABSTRACT

It is becoming more difficult to distinguish artificial intelligence (AI)-generated language from human-written text due to its growing realism. This project targets the classification of fake text using deep learning techniques and assesses three main frameworks—PyTorch, KerasNLP, and JAX—under a unified model architecture. A dataset with labelled samples from AI and human sources is used to test each framework. The same preprocessing procedures, training setups, and assessment criteria are applied when comparing models. KerasNLP outperformed the other two in terms of accuracy and precision. The results provide useful information about choosing a framework for automated text verification tasks.

***K*eywords** Fake Text Detection · AI-generated Text · Deep Learning · Natural Language Processing · KerasNLP · TensorFlow · PyTorch · JAX · Text Classification · Model Comparison · Synthetic Text · Machine Learning Frameworks

1 Introduction

Large-scale language models have advanced to the point where AI-generated text often appears indistinguishable from human writing. This development brings up issues related to automated content creation, academic integrity, and disinformation. As a result, identifying fake or artificial text has grown in importance in natural language processing.

This project explores deep learning-based classification of AI-generated text, with a focus on comparing the performance of different machine learning frameworks rather than designing new model architectures. The study implements models using KerasNLP, PyTorch, and JAX[1, 2, 3, 4], while keeping the neural network structure, data preprocessing, and evaluation criteria constant. This approach separates the framework's contribution to model usability and performance.

The project intends to assist researchers and developers in selecting appropriate tools for tasks pertaining to text authenticity by comparing the outcomes across frameworks. Although each framework had unique benefits in terms of implementation and computational handling, the KerasNLP model performed the best consistently across a number of metrics.

[*] *Citation*: **Yedlapati, S. V. K., Somuri, R., Mengji, D., Palle, P. R. Comparative Analysis of Deep Learning Frameworks for AI-Generated Fake Text Detection. pp. 1–5. DOI: 10.0000/11111.**

2 Project Description

2.1 Objective

The objective of this project is to classify AI-generated text using deep learning and evaluate how different frameworks influence the effectiveness and practicality of the solution. Instead of focusing on model innovation, this study emphasizes a framework-level comparison by using an identical LSTM-based architecture across **KerasNLP** [4], **PyTorch** [2], and **JAX** [3].

2.2 Framework Comparison Rationale

Differences in framework design effect the training speed, ease of debugging, and scalability across systems [4, 5]. This analysis applies an experimental setup to uncover framework-specific trade-offs. It emphasizes on text classification, with a focus on reproducibility and practical deployment considerations [6, 7].

2.3 Model Design and Implementation

To ensure uniformity, all frameworks use the same LSTM-based model architecture. There are three main steps to the process:

- **Input processing**: The native pre-processing capabilities of the corresponding framework are used to clean, tokenize, and encode raw text into numerical sequences.
- **Architecture**: The model includes an embedding layer, one or more LSTM units [8], and concludes with a dense layer employing a sigmoid activation function to perform binary classification.
- **Training setup**: Training utilizes binary cross-entropy as the loss function [9], the Adam optimizer [10], and an 80:20 split between training and validation data.

This design isolates the framework as the only variable affecting performance outcomes by keeping the architecture and training methodology uniform across frameworks.

2.4 Dataset and Preprocessing

The classification task is supported by a balanced dataset, comprising equal quantities of human-written and AI-generated texts [11, 12]. Preprocessing is done by standardizing text by removing irrelevant characters and converting to lowercase. Each framework applies its own tokenizer and vectorizer to convert text into numerical sequences. These sequences are then padded or reduced to a fixed length and grouped into batches for efficient model training.

2.5 Real-World Relevance

When deploying deep learning solutions, framework selection often hinges on practical factors such as compatibility with existing toolchains, availability of hardware acceleration, and the development experience [1, 2, 3, 13]. This project incorporates such considerations by evaluating how each framework manages a realistic natural language processing task—detecting AI-generated content—without introducing architectural discrepancies.

2.6 Contribution

This work provides a reproducible evaluation of deep learning frameworks applied to a text classification problem, independent of specific model variations. It responds to a gap in existing research, where framework selection is frequently overlooked, despite its considerable impact on performance, scalability, and deployment outcomes [14].

3 Our Approach

3.1 Why Compare Frameworks?

In this project, our focus was on evaluating and comparing the performance of various deep learning frameworks—KerasNLP[4], PyTorch[15], and JAX[3]—for the task of AI-generated fake text detection. A key methodological decision we made was to keep the model architecture consistent across all implementations.

3.1.1 Fair Comparison Under Practical Constraints

By maintaining a consistent architecture across all frameworks, we controlled for model-related variations and ensured that observed differences were solely due to the frameworks themselves. This gave us a fair way to measure each framework's performance, ease of use, and ability to optimize. If we had allowed changes to the model architecture, it could've introduced too many variables—making it harder to tell what was really causing any differences in performance.This approach also mirrors real-world scenarios, where developers typically choose frameworks based on deployment requirements, hardware compatibility, or integration with existing systems—rather than architectural differences. In production, models like Transformers or LSTM[8]s are commonly used across frameworks, so our approach aligns with practical use-case decision-making.

3.1.2 Framework Ecosystem, Reproducibility, and Usability

Each framework offers unique tools, APIs, and optimization techniques (e.g., eager execution in PyTorch, XLA in JAX, TensorBoard in TensorFlow), which directly affect training performance, debugging experience, and resource utilization.By using the same model architecture across the board, we made it easier to compare how each framework performs in a clear and consistent way. This setup also boosts reproducibility—future researchers can tweak the framework-specific settings or parameters without having to change the model itself. It not only makes our results more transparent and trustworthy but also easier for others to build on.

3.1.3 Enabling Informed Decision-Making

We weren't trying to pick the "best" model—instead, our goal was to share useful insights into how different frameworks behave when tested under the same conditions. This helps developers make informed decisions when selecting a framework based on their specific project goals, constraints, and technical needs.

3.2 How Our Work Differs from Existing Studies

3.2.1 Framework-Centric Focus over Architectural Innovation

Most existing research in fake text detection emphasizes developing or fine-tuning novel architectures—such as transformer variants or pretrained language models. In contrast, our work keeps the model architecture constant and instead evaluates the frameworks themselves (KerasNLP, PyTorch, and JAX)[3, 2, 4]. This allows us to isolate and analyze the practical impact of each framework in terms of training efficiency, ease of use, scalability, and performance—offering a unique and underexplored angle that is highly relevant for real-world implementation.

3.2.2 Consistent Experimental Design for Fair Comparison

Unlike studies that compare different models or vary multiple experimental variables, we adopt a strictly controlled setup—same dataset, same preprocessing, and the same architecture across all frameworks. This consistency ensures a fair and interpretable comparison, enabling us to draw valid conclusions about each framework's effectiveness. Such a unified evaluation approach is rare in existing literature and directly supports reproducibility and practical decision-making.

4 Data Collection and Pre-processing

4.1 Dataset Description

For this project, we used a publicly available dataset designed for AI-generated text classification. The data set [11][12] comprises two classes: human-written text and machine-generated text. It is appropriate for training and evaluating false text detection systems since it contains samples generated by advanced language models. Supervised learning is made possible by the appropriate labeling of each text sample.

4.2 Data Loading

The data set was loaded using Python and relevant data handling libraries such as Pandas and TensorFlow. To ensure uniformity among samples, the files were read into Data Frames with standardized columns. To make it easier to evaluate the model at various points in time, the data were then divided into training, and validation sets. We confirmed the balance of the classes in each split to avoid any initial bias that could affect the training process [1].

4.3 Data Cleaning

We performed basic text cleaning steps to ensure uniformity. This included:

1. Removal of unwanted characters and excessive whitespace.
2. Lower-casing all text to reduce vocabulary size.
3. Eliminating empty or null samples.

These preprocessing steps help reduce noise and improve the quality of the inputs fed to the models [14].

4.4 Tokenization and Vectorization

We used framework-specific tokenizers for each model implementation. For KerasNLP and TensorFlow models, we utilized kerasnlp tokenizers [4]. The tokenizers convert raw text into sequences of integers representing word or subword units. To maintain consistency across experiments, we fixed the maximum sequence length, used padding and truncation strategies to manage variable length inputs.

4.5 Label Encoding

The binary labels (real vs. fake) were encoded into integers (e.g., 0 and 1) for compatibility with classification models [2]. These were later converted to tensors and one-hot encoded where required.

4.6 Dataset Batching and Preparation

To make it easier for the models to learn from, we preprocessed the data and then grouped it into smaller, more manageable groups called batches. Because processing the entire dataset at once can be computationally demanding, this phase is crucial. To stop the models from picking up any patterns based on the order in which the text samples appear, we also rearranged the data. By supporting improved generalization and training stability, this method enables the models to function well on unknown data.

5 Model Description and Implementation

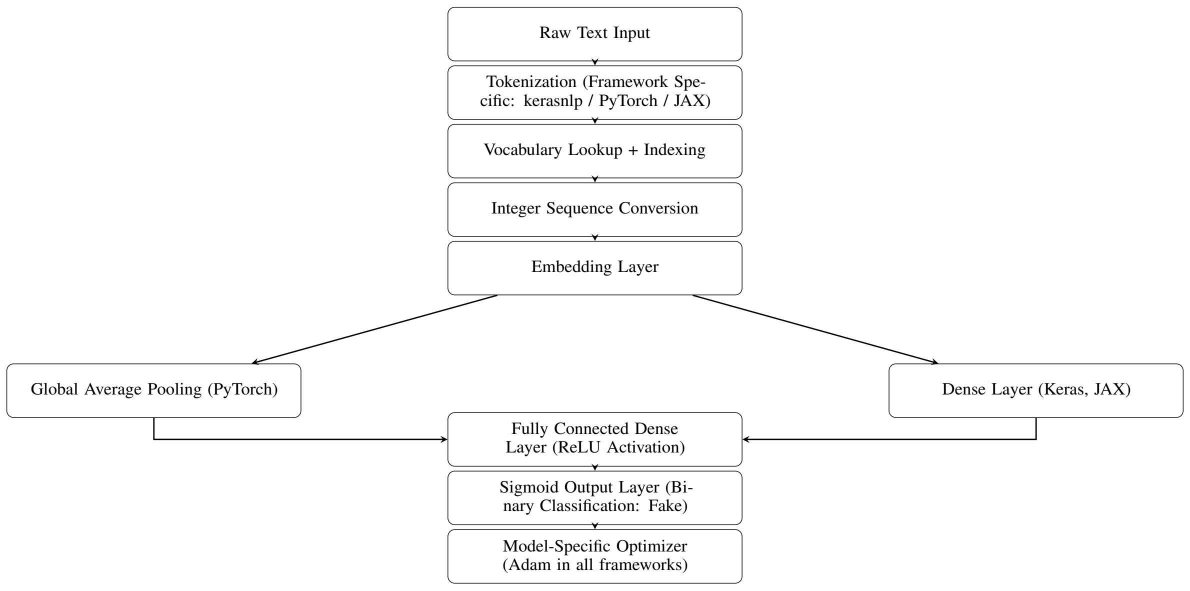
Raw Text Input
Tokenization (Framework Specific: kerasnlp / PyTorch / JAX)
Vocabulary Lookup + Indexing
Integer Sequence Conversion
Embedding Layer
Global Average Pooling (PyTorch)
Dense Layer (Keras, JAX)
Fully Connected Dense Layer (ReLU Activation)
Sigmoid Output Layer (Binary Classification: Fake)
Model-Specific Optimizer (Adam in all frameworks)

Figure 1: LSTM Model Architecture Across Keras, PyTorch, and JAX

5.1 PyTorch Implementation

PyTorch [2] developed by Meta is a popular open source deep learning framework. It is an easy- to-use framework known for its efficiency and flexibility in creating machine learning that can be utilized in wide range of applications ranging from Natural Language Processing, Reinforcement Learning, to Computer Vision. This ability to aid users in the building and training of neural networks makes it extremely viable for research purposes. It has extensive library support and the capability to integrate with other Python libraries which can be used by researchers to carry out multiple collaborative tasks such as data collection and preprocessing, developing and training neural networks.

In this project, we performed multiple data processing steps before developing the implementation of the PyTorch model. The model was built using the core library torch [13] in Python. The torch.nn module in this library is crucial to building the model, since it contains a wide range of modules for defining neural networks. LSTM was used as the preferred type of neural network throughout this project due to its ability to work extremely well with sequential data, remember past words with its memory, and capture the meaning of phrases such as "not bad". The developed PyTorch model contains an embedding layer, a global average pooling layer which helps averaging the output of the embedding layer, and a network of fully connected layers with ReLU activation. The ultimate layer is a sigmoid function layer that classifies the output into two different classes. Hyperparameters such as the maximum sequence length, batch size, and learning rate were modified to achieve the maximum efficiency and improve the overall performance of the model. Data, after data processing and preparation, are differentiated using binary labels to label human-written and AI-generated samples. Label '0' was assigned to human-written texts, while label '1' was assigned to AI-generated texts. Upon labeling the samples, the vocabulary [15] was constructed. This was performed by calculating the word frequencies in the texts and assigning unique integer indices to each word. The vocabulary building helped in tokenizing the input samples into sequences of integers. This proved to be an effective method for classification. The training data was split into sets of 80:20, where 80 was for training the model and 20 was for performing validation tests to evaluate the trained model's accuracy. Binary cross entropy [9] was the chosen loss function that helps optimize the loss.

5.2 Keras Implementation

Keras is a high-level deep learning framework that simplifies the building and training of neural networks. It runs on top of TensorFlow and offers an easy-to-use, modular API for rapid model development. This project involved the usage of modules from Keras and KerasNLP libraries to develop a deep learning model to perform text classification, distinguishing human-written texts from AI-generated texts.

In this project, the preprocessing pipeline in building this model involved tokenization of the data and text preprocessing layers. These were implemented through KerasNLP, ensuring accurate tokenization and input formatting which perfectly aligned with transformer-based architectures [4, 16]. The architecture of this model was built using the `keras.Model` API involving various levels of layers. LSTM was used as the neural network in the Keras implementation. The input layer was followed by an embedding layer that was trainable based on user requirements. This was followed by a stack of transformer encoder blocks that helped capture the contextual dependencies of the input samples [17]. The final layer is a dense output layer with sigmoid activation function for binary classification. The model was then compiled with the Keras Adam optimizer [10] and trained using `model.fit()` on an 80:20 train-validation split. The loss function binary cross-entropy [9] was used here to backpropagate the results of each run, and the accuracy of the model was tracked as the primary evaluation metric. Methods such as early stopping based on validation loss and scheduling learning rate were incorporated to improve generalization and maximize training efficiency. Hyperparameter values such as the count of transformer layers, the size of embedding vectors, and the learning rate were all fine-tuned and set by performing multiple trials and adjustments based on prior results.

5.3 JAX Implementation

JAX [3] is a library developed by Google that provides great performance in computing applications. It supports automatic differentiation, JIT compilation, and vectorization. In addition to that, features such as TPU and GPU acceleration make JAX a valuable tool for machine learning and research.

For this project, the Flax library [18] in JAX [5] was used to build a neural network for binary classification of texts generated by humans or AI. The data preprocessing pipeline included tokenization of the text and the construction of vocabulary from the frequency of words in the input texts, followed by encoding the inputs into definitive length token sequences using padding and truncation [19]. We used LSTM as the neural network in the JAX implementation. The neural network model comprises an embedding layer followed by a network of stacked LSTM units.

The final dense layer was a sigmoid activation that ensures an accurate classification. Hyperparameters such as embedding size, hidden layer dimensions, and learning rate were selected and tuned based on standard practice and empirical tuning. The data set was divided into an 80:20 ratio for training and validation. The training was performed by using the Optax Adam optimizer, with JIT compilation to enhance the performance of the model.

6 Evaluation Metrics

In this project, we used four main evaluation metrics to measure how well our models perform: accuracy, precision, recall, and F1-score[6, 7]. Accuracy tells us how many predictions the model got right overall. Precision shows how many of the texts that the model flagged as fake were actually fake—so it reflects how careful the model is with positive predictions. Recall tells us how many of the actual fake texts the model was able to correctly catch. And the F1-score balances both precision and recall, giving us a single number that reflects both how accurate and how complete the model's predictions are. Together, these metrics help us understand not just whether a model is working, but how well it handles the challenges of fake text detection.

7 Results

In this project, the testing was performed on the AI-GA dataset [12]. This dataset contains samples taken from various AI resources such as GPT-2, GPT-3 and ChatGPT, and human written samples. Another important feature of this data set is that it contains equal numbers of real and fake texts, ensuring fair and transparent comparison of the models. The dataset contains a total of 14331 real samples and 14331 fake samples. This balance in the test dataset is advantageous as it ensures that the performance metrics used in this project, i.e., accuracy, precision, recall, F1 score, are not impacted by class imbalance. Models trained on such a dataset can be evaluated fairly due to their ability to distinguish between fair and real texts. Similarly to the training dataset, the samples from this test dataset were also tokenized into tokens of uniform length and labeled to represent each of the two classes, where '0' was used to represent human-written samples and '1' to represent AI-generated samples, before being used to test the trained models.

7.1 PyTorch

PyTorch Implementation performs moderately with an accuracy of 66% but shows significant confusion of the algorithm to distinguish fake texts from real texts. Its precision for real text detection is 65. 45%, whereas for the detection of fake text, is 66. 55%. This shows that it performs slightly in detecting Fake Text. It shows a high false positive rate (5127 FP) 2, which could be problematic for detecting misinformation. Similarly, it also shows a high false negative rate (4627 FN), indicating that it misses the real text by wrongly labeling it as fake. Its recall for real text is 67. 71% which is better than 64.23% recall for Fake Text. This shows that the PyTorch model shows similar ability in correctly classifying texts into their individual classes.

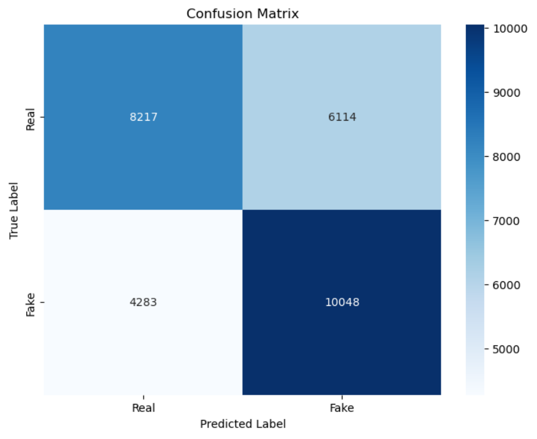

Figure 2: Confusion Matrix - PyTorch Implementation

7.2 Keras

Keras Implementation performs the best among the three models, boasting an accuracy of 70.3%. Its precision for real text detection is good with 67.18% but it really stands with its ~75% precision for Fake Text which is the highest precision among all the implementations. This shows that it performs much better in detecting Fake Text. It shows a high false positive rate (5576 FP) that could be problematic to detect misinformation. However, it shows a much lower false negative rate (2924 FN), indicating that it is strong in detecting real text. Its recall for Real Text is the highest with $\tilde{8}0\%$ but performs bad with 61.1% recall for Fake Text. This shows that though the Keras model can classify the real texts correctly among all the real texts, it is poor at classifying the fake texts correctly.

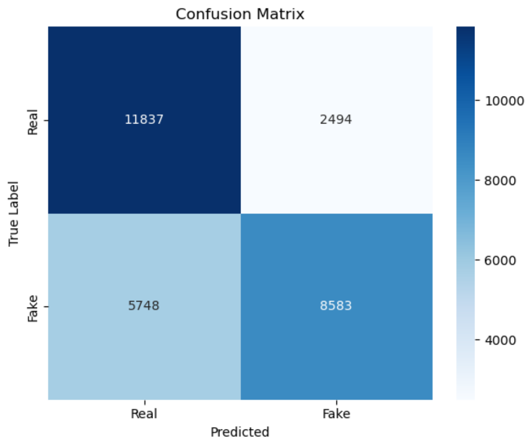

Figure 3: Confusion Matrix - Keras Implementation

7.3 JAX

JAX Implementation performs the worst among the three models with an average accuracy of 64.6%. Its precision for real text detection and fake text detection is close, being 66. 3% and 63. 2% where 63. 2% is the lowest fake text detection score among all. This shows that this model is not good at detecting Fake Text compared to Real Text. It shows a lower false positive rate (4335 FP) compared to a very high false negative rate (5814 FN), indicating that it misses more actual positives. Its recall for Real Text is the lowest with 59.4% but does better with 69.8% recall for Fake Text. This shows that though the JAX model can classify fake texts correctly among all the fake texts, it is poor at classifying the fake texts correctly.

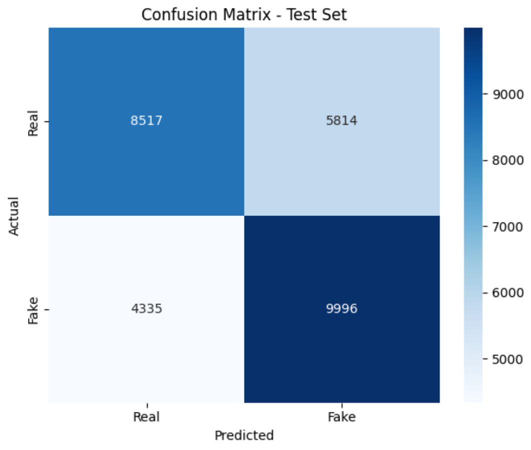

Figure 4: Confusion Matrix - JAX Implementation

The table below 1 presents a clear summary of the evaluation metrics: accuracy, precision, recall, and the F1 score for each framework implementation (Keras, JAX, and PyTorch). This table enables a straightforward comparison of the model performance across all tested frameworks.

Framework	Accuracy	Precision	Recall	F1 Score
Keras	70.34%	0.71 (avg)	0.70 (avg)	0.70 (avg)
JAX	65.00%	0.65 (avg)	0.65 (avg)	0.64 (avg)
PyTorch	65.97%	0.6655	0.6422	0.6536

Table 1: Framework Performance Comparison

8 Team Member's Contributions

1. **Sai Vamsi Krishna Yedlapati** : Took the lead in implementing the model using PyTorch, was responsible for handling the training pipeline, and conducting the initial evaluation. And also worked on creating visualizations such as the confusion matrix for the PyTorch model, and contributed to the model description, results section of the report.
2. **Rohitha Somuri** : Worked on data preprocessing, cleaning, and tokenization for the entire project, ensured the data pipeline was consistent across frameworks and helped with dataset splitting, batching, and label encoding. Also ensured proper dataset handling, padding, and sequence encoding overall. In addition, contributed to the Our approach, Data processing sections in the report.
3. **Dyuti Mengji** : Led the KerasNLP model development, handled model training, and performance tracking and worked on creating visualizations such as the confusion matrix for the KerasNLP model. In addition, contributed to the Project Description, Evaluation Metrics sections in the report.
4. **Pranay Reddy Palle** : Led the JAX implementation, model design, optimization using JIT compilation and the Optax Adam optimizer. Responsible for implementing the LSTM architecture in JAX and vocabulary encoding specific to the Flax stack. Also contributed to drafting the Abstract, Introduction, and Conclusion sections, focusing on framing the motivation and comparative objectives of the study.

9 Conclusion

In conclusion, we used three well-known deep learning frameworks—KerasNLP, PyTorch, and JAX—to create and assess LSTM-based models for identifying AI-generated text. We made sure that their classification performance could be fairly compared by keeping the architecture and preprocessing the same for all implementations. While PyTorch and JAX showed good but marginally worse results, KerasNLP was the model with the highest accuracy and precision. These results show that although all frameworks are capable, their efficacy differs according to computational efficiency, ease of use, and optimisation tools. This information is helpful in choosing the best framework for comparable NLP tasks.

10 Future Work

More complex neural architectures, like transformer-based encoders like BERT or RoBERTa, bidirectional LSTMs, and GRUs, can be implemented and benchmarked in future studies to evaluate their relative efficacy in fake text detection across frameworks. Contextual understanding may be improved by combining pre-trained language models with fine-tuning techniques, particularly for minute differences in generated content. Furthermore, using model interpretability strategies like saliency mapping, attention visualization, or SHAP values may help improve comprehension of model choices, especially in high-stakes scenarios. Metrics like inference latency, memory footprint, and hardware utilization are used to assess deployment efficiency, and ablation studies are carried out to isolate the effects of particular hyperparameters. Lastly, adding multilingual or domain-specific corpora to the dataset (such as legal, medical, or social media content) would enable a more thorough assessment of the model's generalization and resilience in identifying synthetic text in a variety of linguistic contexts.

Acknowledgments

We would like to thank our course instructor and teaching assistants for their continuous support and guidance throughout the development of this project. We also acknowledge the use of publicly available resources and tools that have assisted us so far. All group members have contributed equally to the work done up to this stage, including research, implementation, and documentation, and we will continue to collaborate actively as we move forward with the remaining phases of the project.

References

[1] Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. `https://www.tensorflow.org/`.

[2] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library, 2019.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. Jax: composable transformations of python+numpy programs, 2018. `https://github.com/google/jax`.

[4] Keras Team. Kerasnlp, 2023. `https://keras.io/keras_nlp/`.

[5] Google DeepMind. Using jax to accelerate our research, 2020. `https://deepmind.google/discover/blog/using-jax-to-accelerate-our-research/`.

[6] David Martin Powers. *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation*, volume 2. Journal of Machine Learning Technologies, 2011.

[7] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Aporia. A practical guide to binary cross-entropy and log loss, 2025. Accessed: 2025-05-03.

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

[11] Kaggle Competition Dataset. Llm - detect ai generated text, 2023.

[12] AI-GA Dataset. Ai-ga dataset, 2023. `https://github.com/panagiotisanagnostou/AI-GA/`.

[13] PyTorch Team. Pytorch documentation, 2025.

[14] S. S. Medavarapu. Advancements in deep learning: A review of keras and tensorflow frameworks. *Journal of Scientific and Engineering Research*, 11(5):282–286, 2024.

[15] H. Huang. Text classification with the torchtext library, 2023.

[16] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Sharan Narang, Hongkun Wang, Jaesung Chung, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[17] Ashish Vaswani et al. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[18] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2020.

[19] Phillip Lippe GDE Amsterdam. Introduction to jax with flax, 2020.