Deep Learning for Molecular Biology: Predicting Antibiotic Resistance

Rassul Amantay, Phillip Wollschläger

Summer Semester - 2025

Abstract

Antibiotic-resistant bacteria, particularly *Staphylococcus aureus*, represent a major global health threat. Current resistance detection methods are often time-consuming, resulting in delays in administering effective treatments. Genomic sequencing provides a faster alternative by generating large datasets that are well-suited for analysis using advanced computational approaches such as deep learning.

This study explores the use of deep learning models to predict cefoxitin resistance in *S. aureus* by analyzing the genetic sequence of the *pbp4* gene, a known marker of resistance.

Three types of Recurrent Neural Networks were developed and evaluated: a Simple RNN, a Long Short-Term Memory network, and a Bidirectional LSTM network. These models were trained on *pbp4* gene sequences. Prior to training, nucleotide sequences were numerically encoded and padded to a uniform length. A substantial class imbalance-characterized by an overrepresentation of resistant samples-was mitigated using class weighting, assigning greater importance to the minority non-resistant class. Model performance was evaluated using metrics appropriate for imbalanced data, including F1-score, precision, and recall, as accuracy alone was found to be insufficient. AI assistance was also employed during model development and report preparation.

The results demonstrate the potential of RNN-based deep learning models for rapid prediction of antibiotic resistance from genomic data, provided that class imbalance is appropriately addressed. In this study, model performance was primarily limited by dataset size rather than model architecture. The Bidirectional LSTM is recommended for future work involving larger and more complex datasets, due to its theoretical advantages in sequence modeling.

Contents

1 Introduction

The rise of antibiotic resistance is a growing problem in modern medicine. Bacteria like *Staphylococcus aureus* are especially concerning because they can become resistant to several antibiotics, making infections harder to treat. To deal with this issue, it is important to quickly and accurately find out if a bacterial strain is resistant to a certain drug.

Traditional lab tests for detecting resistance can take a long time. On the other hand, genomic sequencing allows faster access to genetic data, which opens the door to using advanced computer methods like deep learning to analyze this information.

This report describes how deep learning was used to predict whether *S. aureus* is resistant to the antibiotic cefoxitin. The prediction is based on the DNA sequence of the *pbp4* gene, which is linked to resistance. The main goal was to build and test different Recurrent Neural Network (RNN) models: a Simple RNN, a Long Short-Term Memory (LSTM) network, and a Bidirectional LSTM (Bi-LSTM) network. Since the dataset had many more resistant samples than non-resistant ones, special steps were taken to reduce this imbalance and improve model fairness.

The work followed an iterative process-starting with a simple model and moving to more complex ones-to see which one worked best. The models were compared based on how well they performed and how suitable they are for this kind of genetic data analysis.

2 Theoretical background: Recurrent Neural Networks

At the core of this prediction system are RNNs, a type of neural network designed to work with sequences, where the order of elements matters. Unlike standard neural networks, RNNs have a built-in memory that helps them keep track of earlier parts of a sequence. This makes them useful for tasks like time-series forecasting, language processing, and, in this case, working with DNA sequences.

2.1 Simple Recurrent Neural Network

The Simple RNN is the most basic form of this network type. It reads a sequence step by step, updating its memory (called the hidden state) each time. This lets the model learn from earlier elements in the sequence. However, Simple RNNs have a known limitation called the **vanishing gradient problem**. During training, the signal used to adjust weights can get very small as it moves backward through the sequence. This makes it hard for the model to remember information from earlier in long sequences. In the case of DNA, it means the model might forget important patterns at the beginning of the gene.

2.2 Long Short-Term Memory

LSTM networks were created to fix this memory problem in Simple RNNs. They use a more advanced structure that includes a special memory unit called the cell state. This memory is carefully controlled by three parts known as gates: the forget gate, which decides what old information to throw away; the input gate, which decides what new information to add; and the output gate, which decides what to send out to the next step. Thanks to this system, LSTMs can remember useful information over long sequences. This makes them better for working with full gene sequences, where patterns far apart may still be related to resistance.

2.3 Bidirectional LSTM

While a regular LSTM reads a sequence from start to end, a Bi-LSTM processes it in both directions-forward and backward-at the same time. It uses two separate LSTM layers: one reads from the beginning to the end, and the other from the end to the beginning. Their outputs are then combined to give a fuller view of the sequence. This approach helps the model understand each element not just based on what came before, but also on what comes after. In DNA analysis, where the role of a nucleotide can depend on its neighbors on both sides, this extra context can improve performance.

3 Materials and methods

3.1 Dataset

The main data came from the **seminar-dlmb-2024-winter-public** repository. It contains *pbp4* gene sequences from *Staphylococcus aureus* along with labels showing whether the bacteria are resistant to the antibiotic cefoxitin (0 means not resistant, 1 means resistant). One important feature of this dataset is that there are many more resistant samples than non-resistant ones.

3.2 Data preprocessing

To prepare the DNA sequences for the neural networks, each nucleotide was converted into a unique numerical format using one-hot encoding: 'A' became [1,0,0,0], 'C' became [0,1,0,0], 'G' became [0,0,1,0], and 'T' became [0,0,0,1]. Any unclear characters or padding were represented by [0,0,0,0]. Because the sequences have different lengths, all sequences were padded with these zero vectors so they all have the same length as the longest sequence in the training set. This uniform shape is necessary for feeding data into the models.

3.3 Model architectures

Three types of RNN models were built using the Keras Sequential API. The first, a Simple RNN, had two stacked `SimpleRNN` layers followed by a `Dense` output layer. The second replaced these layers with `LSTM` layers to better handle long-term dependencies. The third and most advanced model used Bi-LSTM layers, which allow the network to read the input sequences both forward and backward at the same time. All models included `Dropout` layers to help prevent overfitting.

3.4 Training and evaluation

Model Compilation and training

All models were compiled with the Adam optimizer and binary crossentropy loss, which fits the task of classifying resistance. To handle the class imbalance, we used class weighting during training, giving more importance to the minority non-resistant class to reduce bias. Models were trained for up to 100 epochs with a batch size of 32. We also used early stopping to stop training if the validation loss did not improve for 10 epochs in a row, helping to avoid overfitting.

Evaluation metrics

Because the dataset is imbalanced, accuracy alone can be misleading. So, we evaluated models using metrics based on the confusion matrix: True Positives (TP), True Negatives (TN), False Positives (FP), and False

Negatives (FN). These represent correctly and incorrectly classified resistant and non-resistant samples.

From these, we calculated:

- **Precision**, which shows how many of the predicted resistant samples were actually resistant.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall** (or sensitivity), which shows how many of the actual resistant samples were correctly identified.

$$Recall = \frac{TP}{TP + FN}$$

- **F1-Score**, which combines precision and recall into one measure, balancing both and useful especially for imbalanced data.

$$F1\_Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

These metrics were calculated for both the resistant and non-resistant classes. To summarize the overall performance in the final classification report, we use two different averaging methods:

- **Macro Average:** This calculates the metric independently for each class and then computes the simple average. It treats both the resistant and non-resistant classes as equally important. This is a crucial measure for our study, as it provides an unbiased view of the model's ability to handle the rare non-resistant class.
- **Weighted Average:** This also calculates the metric for each class but then computes an average that is weighted by the number of samples in each class. This score reflects the model's expected performance on the dataset as a whole but is naturally influenced more by the performance on the more common resistant class.

3.5 Software and libraries

All computational experiments were performed using Python 3.12.7. Key libraries and their versions include:

- **TensorFlow:** 2.16.1
- **Keras:** 3.3.3
- **NumPy:** 1.26.4
- **Scikit-Learn:** 1.4.2
- **Matplotlib:** 3.8.4

3.6 AI-assisted methodology

During this project, an AI assistant was used to help with debugging and clarifying complex concepts in deep learning and biology. The AI helped explain error messages and suggested possible fixes, which sped up the troubleshooting process. However, all the core work, including writing the code, designing the models, analyzing results, and drawing conclusions, was done by us. All final code and text were carefully checked and approved by us.

4 Results

4.1 Initial findings and the challenge of data imbalance

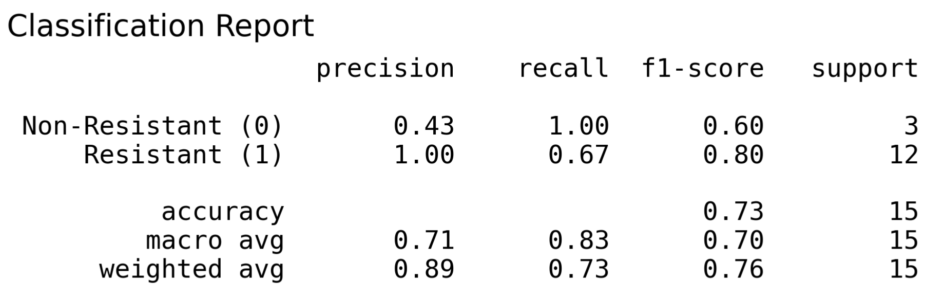
Initial experiments involved training three recurrent neural network architectures - Simple RNN, LSTM, and Bi-LSTM - on the provided dataset. These models consistently achieved a test accuracy of approximately 80%. However, a deeper analysis of the dataset revealed a significant underlying problem: a severe class imbalance. The training data consisted of 135 sequences, with 98 labeled as resistant (73%) and only 37 as non-resistant (27%). The test set was even more skewed, including 12 resistant samples and only 3 non-resistant samples, reflecting an 80/20 split. This imbalance led to a misleading accuracy score, as the models had learned a simple, biased shortcut by predominantly guessing the majority class - resistant. The initial high accuracy was therefore not an indicator of a successful model, but rather an artifact of a flawed dataset.

4.2 Corrective measures: implementing class weighting

To address this bias and force the models to learn the actual distinguishing features in the DNA, a standard and effective technique, **class weighting**, was implemented. By applying weights inversely proportional to class frequencies during training, the model was penalized more heavily for misclassifying the rare non-resistant class. This compelled the models to pay significantly more attention to these crucial minority samples.

4.3 Post-correction model performance

After implementing class weighting, the models were retrained and re-evaluated. The results were dramatic and insightful. The most significant finding was that all three architectures - Simple RNN, LSTM, and Bidirectional LSTM - converged to the exact same performance metrics and confusion matrix (Figure 1).



```
Classification Report
                    precision    recall  f1-score   support

Non-Resistant (0)       0.43      1.00      0.60         3
    Resistant (1)       1.00      0.67      0.80        12

         accuracy                           0.73        15
        macro avg       0.71      0.83      0.70        15
     weighted avg       0.89      0.73      0.76        15
```

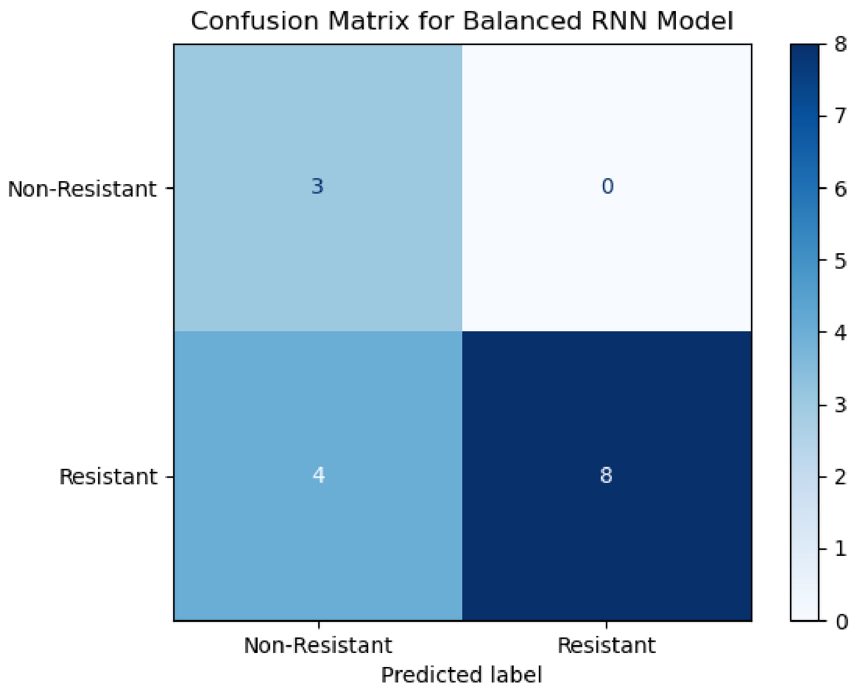


Figure 1: Classification Report and Confusion Matrix for all three implemented architectures.

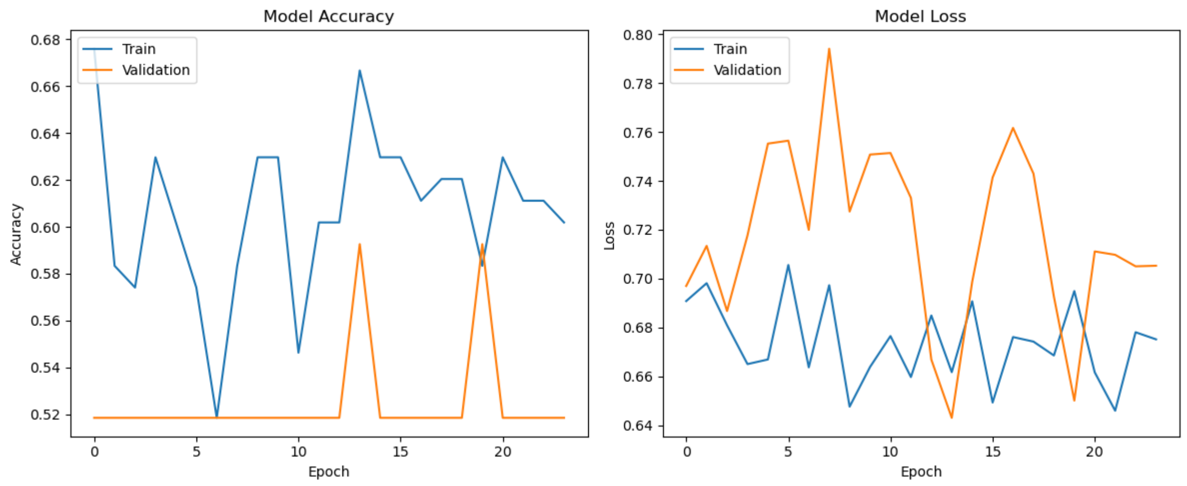


Figure 2: Training History of the Simple RNN Model (Representative of all three architectures). The left plot shows the model accuracy, and the right plot shows the model loss, both on training and validation datasets across epochs.

Key performance metrics for the non-resistant class (minority) included a recall of 1.00, precision of 0.43, and an F1-score of 0.60. For the resistant class (majority), the recall was 0.67, precision 1.00, and the F1-score 0.80. The most critical result is the recall of 1.00 for the non-resistant class, indicating that the balanced models successfully identified every non-resistant sample in the test set, completely overcoming the initial problem where these samples were ignored. The trade-off was a reduction in recall for the resistant class, as the model no longer defaulted to this prediction, demonstrating a successful shift from a biased, naive model to one engaged in genuine pattern recognition.

5 Discussion

5.1 Overfitting, underfitting, and model complexity

A primary objective during model development was to create a generalizable model by mitigating overfitting. Standard regularization techniques were employed, including Dropout layers to prevent neuron co-adaptation and EarlyStopping to halt training when validation performance plateaued.

However, a critical analysis of the training history (Figure 2) reveals that these measures were insufficient to fully prevent overfitting. The clear divergence between the steadily decreasing training loss and the highly volatile, non-improving validation loss is a classic indicator that the models began to memorize the training data rather than learning its underlying patterns. This behavior is a direct consequence of the dataset's small size. The models were complex enough to quickly learn the limited genuine signals present and subsequently began fitting to the noise, a common pitfall when training deep learning models on scarce data.

5.2 Interpretation of identical performance

All models reaching the same outcome suggests that the dataset was not complex enough to need the advanced features of LSTM or Bi-LSTM. The simpler RNN was able to find the important patterns, and the more complex models could not improve because there were no additional patterns to discover given the limited data.

5.3 The impact of hyperparameters

To compare models fairly, we used the same set of hyperparameters (like layer sizes, dropout rates, and optimizer) for all models. This helped us focus on the differences between the architectures themselves, but it did not aim to get the best performance for any single model. It is likely that tuning hyperparameters individually for each model could improve results, but this would make more sense if we had a larger dataset with more complex patterns.

5.4 Limitations of the study

The main limitation is the size of the dataset. While it was enough to demonstrate the basic idea and the importance of dealing with class imbalance, the small number of samples-especially in the minority class-limits how well the model can predict. Because of this, the model's ability to generalize to the wider population of *Staphylococcus aureus* strains is restricted by the diversity of the data it was trained on.

6 Conclusion

This project demonstrated the successful development of a deep learning model for predicting antibiotic resistance from gene sequences. Addressing the class imbalance issue proved crucial, allowing the models to accurately identify rare non-resistant cases despite a modest overall accuracy. This highlights the importance of careful data handling in bioinformatics applications.

Although this work concludes here, the results suggest promising avenues for further exploration. Expanding to larger and more varied datasets, alongside fine-tuning model parameters, could unlock greater predictive power and reveal deeper biological insights. Even without immediate continuation, the foundation laid here provides valuable guidance for future studies in this area.

7 Code and data availability

The source code, data, and additional materials for this project are publicly available on GitHub at: `https://github.com/RassulAmantay7/RNN-in-DNA-Sequences`

8 Alternative Experimental Approach

This appendix details an alternative experimental approach that was explored, utilizing the PyTorch framework. This methodology focused on data augmentation to handle class imbalance and the use of k-mer based feature encoding.

8.1 Methodology

The class imbalance was addressed via data augmentation, where samples from the minority class (non-resistant) were duplicated to create a balanced training set. For feature engineering, several encoding schemes were tested, including standard one-hot encoding and k-mer encoding, where sequences are broken into overlapping subsequences of length k.

A systematic grid search was performed to find an optimal model configuration, exploring various combinations of hyperparameters. The top-performing models from this search were then selected for final evaluation. This experimental track was conducted using Python 3.13.5, PyTorch 2.7.1, NumPy 2.3.1, and Biopython 1.8.0.

8.2 Results

The final models were evaluated on the original, unaugmented dataset. The performance of the top three models on the test set is summarized in Table 1. The confusion matrices for both training and test sets are visualized in Figure 3.

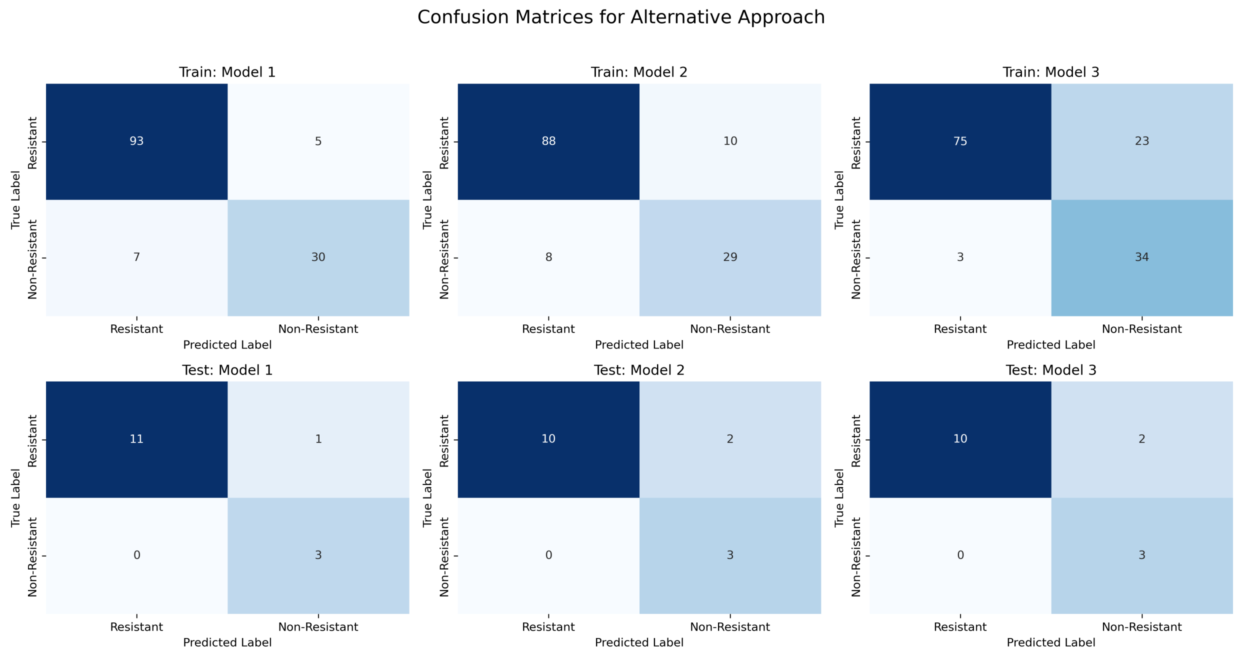


Figure 3: Confusion matrices for the top three models on both the unaugmented training data (top row) and the test data (bottom row).

Table 1: Performance of the top three models from the alternative approach on the test set.

Model ID	**Encoding**	**F1-Score**	**Accuracy**	**Precision**	**Recall**
Model 1	k-mer ($k = 4$)	0.957	0.933	1.000	0.917
Model 2	k-mer ($k = 4$)	0.909	0.867	1.000	0.833
Model 3	k-mer ($k = 3$)	0.909	0.867	1.000	0.833

The models selected through the grid search all utilized k-mer encoding as their feature input.

8.3 Code and Data Availability

The source code and materials for this approach are publicly available at the following repository.

`https://git.rz.tu-bs.de/phillip.wollschlaeger/seminar_hzi_2025`

Permanent link to the specific commit:

`https://git.rz.tu-bs.de/phillip.wollschlaeger/seminar_hzi_2025/-/tree/`
`b344afde5ea7daa7a4a68487d554c99a6d7d8720/`

References

[1] Shen, J., Liu, F., Tu, Y., Tang, C. (2021). Finding gene network topologies for given biological function with recurrent neural network. *Nature Communications*, *12*(1), 3105. https://www.nature.com/articles/s41467-021-23420-5

[2] Lian, X., Zhang, Y., Li, Y., Zhang, Z. (2018). Deep recurrent neural network discovers complex biological rules to distinguish mRNA from lncRNA. *Nucleic Acids Research*, *46*(16), 8105–8115.https://academic.oup.com/nar/article/46/16/8105/5050624

[3] Pearson, W. R. (1994). Using the FASTA Program to Search Protein and DNA Sequence Databases. In: Griffin, A. M., Griffin, H. G. (Eds.), *Computer Analysis of Sequence Data* (pp. 307–331). Humana Press. https://link.springer.com/protocol/10.1385/0-89603-246-9:307

[4] Lee, J. Y., Kim, S. J. (2016). PBP4: A New Perspective on Staphylococcus aureus β-Lactam Resistance. *Antibiotics*, *6*(3), 57. https://www.mdpi.com/2076-2607/6/3/57

[5] University of Waterloo. Lecture on Recurrent Neural Networks. Available at: https://bit.ly/2RCNEhn

[6] Understanding LSTMs. Available at: https://bit.ly/1S6gmjZ

[7] MIT Notes on Recurrent Neural Networks. Available at: https://tinyurl.com/2x4z77fz

[8] Hinton, G. Lecture on Recurrent Neural Networks. Available at: https://bit.ly/3TNBPqw

[9] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Available at: http://www.deeplearningbook.org