Deep Learning Program 1

Arushi Mangla
SCU ID: 07700000114

Title

Deep Learning for Peptide Classification

Abstract

Proteins, or more essentially biomolecules, carry out diverse functions in living organisms, driven by their unique amino acid sequences and resulting three-dimensional structures. Biofilms, dense microbial communities encased in matrices, pose significant health risks due to their resistance to antimicrobials. Identifying peptides with dual antimicrobial and antibiofilm properties is crucial for combatting biofilm-related infections. In this program, I have tried classifying an antibacterial peptide string if it also an antibiofilm or non-antibiofilm. For this classification, I have designed two separate predictive feed-forward neural networks – one using NumPy library in Python and the other one using the Sequential function in Tensorflow.

Introduction

Proteins are essential for every living being on this planet to carry out various life functions inside their body. We can differentiate between types of proteins through their sequence of amino acids where a linear chain of amino acid residues is called a polypeptide and short polypeptides are commonly known as peptides.

Identifying peptides with dual antimicrobial and antibiofilm properties is crucial for combatting biofilm-related infections and to mitigate the risk of such diseases. This report summarizes the approach used for classifying a peptide string as antibiofilm or not using two separate deep learning models.

Dataset

The training and test data used contained a peptide string representing an amino acid residue. The training set also contains the labels indicating the class of peptides and is represented with a value of 1 (antibiofilm) or -1 (non-antibiofilm). In both the data sets, the characters are separated by the delimiter tab. The training data is imbalanced where the minority class is '1' with only 142 instances while the '-1' class has 1424 instances.

	labels	columns
0	-1	DVELDLVEISPNALP
1	-1	KADEELFNKLFFGT
2	-1	FLVALHLGTAFALLWYFRKRWCALVRGFFASFGGRRNDDAHMM
3	-1	RDQMRARIADITGVAISRIA
4	-1	RKRLQLLLL

Figure 1. Train Dataset Preview

Methodology

a. **Data pre-processing**

It is a crucial step before as it helps in improving the performance and accuracy of the trained model, by refining the quality of the input data, normalizing the training attributes and identifying inconsistencies in data.

1. **Feature extraction** - k-mers of length 3 was applied to get short subsequences of characters from the strings. This was done by creating a k-mers function which could generate a list of k-mers for length 1 – (k-1) and then store them in a list. Both the training and test sets were passed into this

function where each string in both the sets was taken and k-mers were extracted. A dense matrix was created.

2. **Oversampling** - Imbalance in the train data was addressed by using the RandomOverSampler function to oversample the minority class.

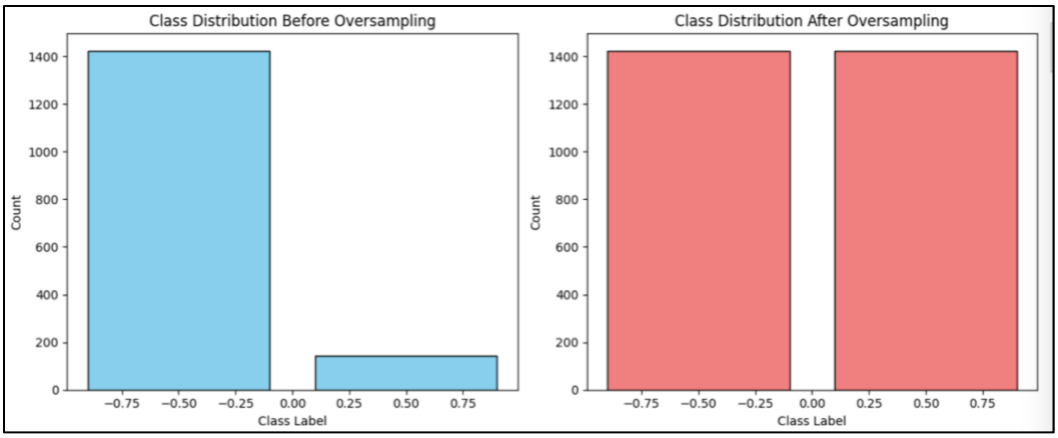

Figure 2. Class Distribution Before and After Oversampling

b. Train and Validation Split

The train data was split into training and validation set in a ratio of 80:20 using the train_test_split function.

c. Neural Network Architecture

1. **NumPy** - A two-layer neural network is created consisting of an input layer of 436 neurons, a hidden layer of 200 neurons (decided after testing with different number of hidden neurons) and one output layer for our binary classification task. ReLU activation function is used in the hidden layer so that it can learn from the complex kmers sequences and to create non-linear decision boundaries for robust classification. This also introduces sparsity which will help the model in focusing on the relevant features and ignore the irrelevant ones, which can be observed in case of character strings. I chose the sigmoid activation function in the output layer as it is efficient when we have to predict the probabilities of belonging to one of two classes. It gives the output in a range of 0 or 1, which is why it works well in case of binary classification.
2. **Tensorflow** - Using a deep learning framework, I have built a sequential model using the tensorflow library. In this model, the architecture consists of a dense layer with 128 neurons, followed by a dropout layer to prevent overfitting, another dense layer with 64 neurons and a batch normalization layer to improve the performance by normalizing the input of each layer. In both the dense layers, I have used a ReLU activation function for robust classification. The final dense layer consists of one neuron and a sigmoid activation function for binary classification. In both the dense layers

Model Training

a. NumPy - Gradient descent has been employed for model training which helps in reducing loss by finding the optimal values for weights and biases. The number of epochs used are 200 and the learning rate is kept at 0.03. The hyperparameters were decided after testing out different values on multiple iterations of the model.

b. Tensorflow - In the tensorflow model, I have used the Adam optimizer for training with a learning rate of 0.001. The number of epochs on which the model was trained were kept 50 since the size of the dataset was very small.

Challenges

a. NumPy - Setting the initial weights for the neural network was a difficult task considering the small size of the dataset. I tried addressing this issue using the gradient descent method but there is scope for it to be optimized further. Moreover, multiple iterations had to be done while designing the neural net to understand an efficient architecture and to set the learning rate at an optimal value.

b. **Tensorflow** - In this model, controlling overfitting was a major issue since the size of the dataset is very small. I added a dropout layer and a batch normalization method to reduce it but I feel it can be improved further.

Results

a. **NumPy** - I achieved an accuracy of 50% on the validation test and MCC score for the test data came out to be 80.93%. The train and validation loss curve is going down which tells me that the model was performing well, however they are too close to each other and can be rectified using more data in the model.

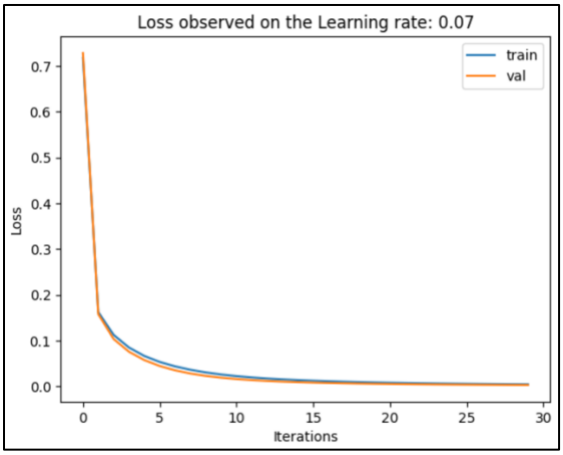

Figure 3. Training and validation loss for NumPy Neural Network

b. **Tensorflow** - I achieved an accuracy of 50.26% with this model which is the same I achieved from the NumPy Neural Network model. The training and validation curve is going down but is not coming to a saturation point which means that the model can be improved further by adding more layers or by using different hyperparameters.

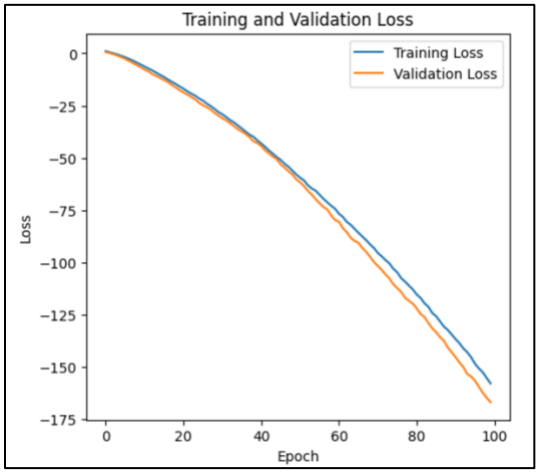

Figure 4. Training and validation loss for Tensorflow Neural Network

References

- [Tensorflow API Docs]
- [Neural Network Implementation using NumPy]