Machine Learning on Protein Structure Prediction

Yanghe Liu
yl4350@nyu.edu

Szumin Yu
smy320@nyu.edu

Haowen Zheng
hz1627@nyu.edu

Abstract

In this project, we applied different machine learning algorithms with a purpose to predict the distance matrix for the test data, which consists protein amino acid sequence. The model was trained on the dataset with 4,454 amino acid sequences. K Nearest Neighbors was used as the baseline standard. In addition, we also tried Long-short Term Memory (LSTM) models and automated machine learning methods.

1. Introduction

This project applies machine learning techniques to the sequence data. Sequence prediction problems have been around for a long time. They are considered as one of the hardest problems to solve in the data science industry.

In this specific case, we were given a training set of 4,454 rows of amino acids sequences. The first was a sequence of letters over 21 alphabets (amino acids), and the second was a sequence of letters over eight alphabets which represented the secondary structure. The training output consisted of 4,454 distance matrices, each having the same number of rows and columns, and corresponds to the length of the sequence. The matrices were symmetric with zeros on the diagonal axis, denoting distances between the amino acids when folded in the specific structures.

We tried three different methods in total, K Nearest Neighbors (which we used as a baseline method), LSTM, and automated machine learning. Under each specific method, we tried many different ways to achieve the optimal results. In this report, we recorded the following aspects, organized by methods. Under each method, firstly, we noted down our research on relevant work, based on which we developed our understanding of the specific method. Then we described the architectures in building our models. In addition, we presented the results we achieved, and finally, we recorded the trail and errors that we encountered in the whole process.

For this project, we ended up with the LSTM model, which achieved the best scores on the leadership board.

2. Problem formation

This project aimed to predict the distance matrices for the new test inputs, which consisted of 2,24 rows of sequences information. It was a supervised machine learning problem, with a training data with sequential feature input and distance matrix as labels. Based on the methods we picked, the problem could be either classification problem or regression problem. For instance, in our baseline method, K Nearest Neighbors Method, we were solving a classification problem if we assigned the label of the nearest training example to the test example as the prediction output (k=1). On the other hand, if we set the k in KNN to be greater than 1, the problem would be a regression problem. For other methods that we tried, like the LSTM (Long-short term memory) model, by which we generated the outputs to be the mean and variance of the distance matrix, we were also solving a regression problem.

3. K Nearest Neighbors

Generally, the k-nearest-neighbors algorithm (KNN) is a non-parametric method used for classification and regression. The input consists of the k closest training examples in the feature space. The output, however, depends on whether KNN is applied for classification or regression. In this project, we treated our problem as classification while using KNN.

When KNN is used for classification, the output can be calculated as the class with the highest frequency from the K most similar neighbors. Each neighbor votes for their class and the class with the most votes is taken as the prediction of the object. If we use k equals to one, then the output will be directly assigned to the class of that single nearest neighbor, because in our case every matrix is different. There are different methods to determine which of the K instances in the training dataset are most similar to the test dataset. For real-valued input variables, the most commonly used measure is Euclidean distance. In our case, we use the longest common sub-sequence(LSC) measure as our measurement. The details about LSC will be explained in the following part.

3.1. Architecture

The first method we tried was the K Nearest Neighbors, where we used $k = 1$. We used K Nearest Neighbors as the baseline method. For each row in the test set, we attempted to find the nearest neighbor in the training input data set. In other words, we wanted to identify a protein with the know distance matrix that most "looked like" the new protein for which we aimed to predict the matrix output, given the information on their sequences. Then we simply assigned the matrix of the identified training example to the test protein.

To achieve this goal, we took two steps.

The first step was for a test input, to identify the proteins that had the same lengths of the sequences. The second step was then to identify among those proteins (selected out by the first step), the one with the longest common sub-sequence.

After a few trials and errors, we found the following method achieved the best output. The other attempts were described in the last subsection "Trials and errors."

Given that the lengths of amino acids and q8 sequences were the same, we first held the test sample to find the training samples with the same length of the sequences. Then we tried to identify the longest common sequence in both amino acids and secondary structure in every training sample with the same length. We then added the lengths of longest common sub-sequence for both sequences (amino acid sequence and q8) up to identify the one (in the training set) with the maximum number. We returned the index of the training example, and finally output the corresponding matrix.

3.2. Results

Figure 1 is a visualization of four output matrices for the test set examples. We used these visualizations to check whether the KNN method achieved at least a baseline standard. As we used the $k = 1$ and were assigning the matrix of the "most looked alike" training example to the test example, we expected to see zeros on the diagonal and symmetry in the matrix. From Figure 1 we saw we had at least met the most basic expectations. However, as we were assigning matrices from the training output to the test output, we also expected to see substantial squared errors when comparing our output to the real test output, the ground truth.

3.3. Trials and errors

Before we arrived with the algorithm described above in the KNN section, we went through a few trials and errors. Here we recorded the attempts that we tried but deserted in the end.

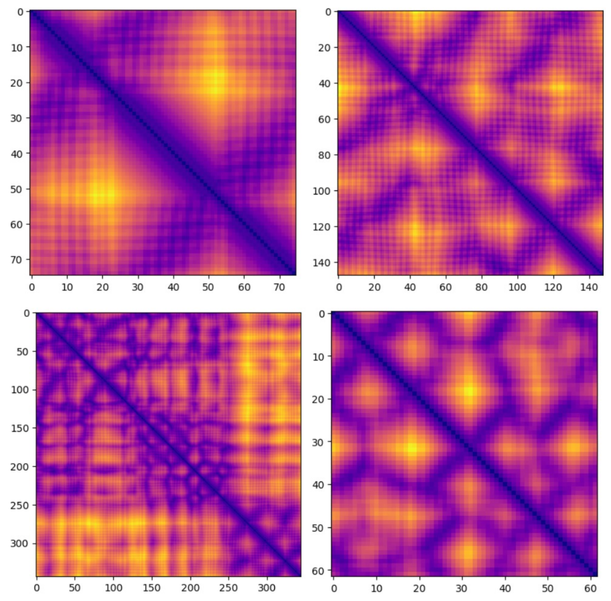
Figure 1. Examples of output matrices by KNN method (k=1).

3.3.1 Concatenating sequences for feature input

Our first attempt in dealing with feature input was to concatenate the two sequences together. We thought this would save space in the Python codes, and as the algorithm did fewer iterations, it was less computation costly. However, we suspected there would be errors. It was possible for the algorithm to match common sub-sequences at different structures. For example, the algorithm could take a sub-sequence from the amino acids for the test example but matched it to a sub-sequence that included both alphabets in the amino acids and the secondary structure. It turned out not an optimal method, so we ended up not using it eventually.

3.3.2 LSC

We also tried different ways to process the longest common sub-sequence (LSC). We did some research and found some people counted the times of specific characters showing up in the sequence, and some extracted the continuous sub-sequences. We tried both ways, and the files were included in Google Drive. We later decided that the model which extracted consecutive sub-sequences be the better method for this project.

3.3.3 Deciding the number of K

We also tried to run a KNN model with $k = 10$ and $k = 4$ by applying the baseline code uploaded to Google Drive.

We discussed the selection of k, and based on our knowledge, when k is small, we would be restraining the region of a given prediction and forcing our classifier to be "blinder" to the overall distribution. A small value for k could provide the most flexible fit, which will have low bias but high variance. On the other hand, a higher k would average more voters in each prediction and hence could be more resilient to outliers. Larger values of k would have smoother decision boundaries which lowered the variance but increased bias.

Ideally, we could perform an elbow check, however, given that time was limited, we ended up picking a slightly larger k number. Although it was computationally heavy, as the example code using $k = 5$ already took a considerable amount of time, we decided to try k = 10, which was bigger than the default (K = 5) since we expected a larger K to produce a smoother decision boundary and alleviate the problem of over-fitting. It turned out to be precisely time costly as we expected, and it took a whole night (more than 6 hours) to run.

However, we found the results were far off our expectation. The RMSE was very large. We then also tried $k = 4$, the result was not too satisfying either. Thus we ended up not using $k > 1$.

3.4. Potential Improvement

There are two methods we think might be able to improve our KNN model. In the KNN model we discussed above, we treat the sequence (both amino acids and secondary structure) as a complete sequence from where we searched for LCS. The first method is to split both training sequences and test sequences to n parts and then compare the sub-sequence of the test data with the sub-sequence of the training data in the n smaller fractions. The output matrix would accordingly be n matrices, which we could do a simple average over. This thought is based on our assumption that it might achieve more accurate results if the sequences are broken smaller so that it could capture more information.

So far, we still only compared the test instances with the training instances which have the same continuous sequence. When we researched KNN with sequential biological data, BLAST came up as a recommended way of finding the longest common sequence in KNN most efficiently. BLAST is one of the most widely used bioinformatics programs for sequence searching [6]. The main idea of BLAST is that there are often High-scoring Segment Pairs (HSP) contained in a statistically significant alignment. Instead of searching for longest common sequence between test sequence and the training sequences with the same length, BLAST algorithm searches for high scoring sequence alignments between the test sequence and all of the training sequences in the database using a heuristic approach. This method should produce a more accurate result than the LCS method we discussed above because it does not only focus on the training data with the same length but squeezes the information from all of the training data. It will be a suitable method for us to improve the KNN model in the future.

3.5. References

Some of the codes we used to find the Longest Common Subsequence are from this website [3].

4. LSTM

From KNN we moved on to the LSTM model (Long Short-Term Memory) proposed by Hochreiter and Schmidhuber in 1997 [8]. The LSTM model has been observed to be one of the most effective solutions for machine learning problems using sequence data, mostly because of its property of being able to remember patterns for a long duration of time selectively. It ameliorates the problem of vanishing gradients, an improvement from the basic RNN (recurrent neural network) models, because simple RNN models have the limitation of not being able to recall information passed along a certain time ago. Hence, simple RNN models have very short "memory." It would lead to a problem of vanishing/exploding gradient.

Like a feed-forward conventional neural network, in simple RNN models, the weight being updated for a particular layer is calculated by multiplying the learning rate, the error term from the previous layer and the input to that layer. In this way, the error term for a particular layer ended up to be a product of all previous layers errors. Because we would use activation functions like the sigmoid function, we could get minimal values of the derivatives. Multiplying them would then lead to the gradient vanishing, causing difficulty in the training of layers.

In our case, a simple RNN model would not take into account a lot of the useful information in the sequences. Instead, we should use a special version of RNN, the LSTM, which makes small modifications to the input information by multiplications and additions. Besides, since our data is amino acid sequences, the order of amino acids might affect one another. Therefore, based on the characteristics of our training data, we used LSTM in our pipeline.

4.1. Basic Architecture for a single LSTM

Here we gave a brief review of a single typical LSTM model, before we continued to describe our construction of mult-layered LSTM model, as it is essential for understanding why the LSTM ended up performing way better than our results from KNN.

Like the image below, a single typical LSTM network consists of different memory blocks which are called the cells, the rectangles that we see in the image [1]. In the cell, the newly input information (x_t) and the output from previous input (h_{t-1}) are manipulated. As illustrated in Figure

2, there are four functioned working for this purpose, and they are grouped as three mechanisms, which are also called "gates." From left to right, the first gate is the forget gate, which takes two inputs, h_{t-1} and x_t. Through a sigmoid function, it output a vector from 0 to 1, which indicated if the two input pieces are to discard or keep. The second gate is the input gate, which involved the second sigmoid function and the tanh function. At this stage, the sigmoid function filtered the information from h_{t-1} and x_t, and the tanh function created a vector containing all possible values that can be added to the cell, from -1 to +1. Then the model multiplies the two output vectors and adds the result to the cell state. The last gate is the output gate, which takes another sigmoid function, and a tanh function. The process is broken up to three steps. First, the sigmoid function filter inputs from h_{t-1} and x_t to regulate values values to output from each input source (the output from the prior state and the current state). Then applying the tanh function, it creates a vector rescaling the values for output. Finally, the algorithm is a multiplication of the value of the regulatory filter and the vector created by tanh.

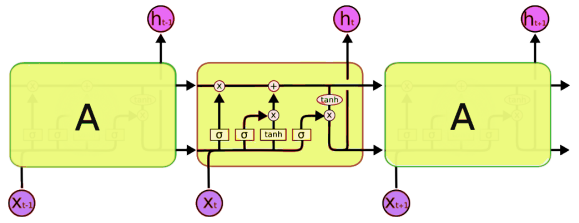

Figure 2. Architecture of a basic LSTM model. Source: Olah, C. *Understanding LSTM Models.* [5].

4.2. Constructing our multi-layer LSTM

After fully grasping the theoretical knowledge of LSTM, we started to build our LSTM model using the library Keras, a high-level API for neural networks which works on top of Tensorflow, for the sake of this course project.

4.2.1 Define the LSTM model

We used a sequential model, where we simply stacked six layers in total.

The first layer is an LSTM layer with 128 memory cells/neurons. We made sure that it returns sequences, so that the next layer could receive sequences instead of randomly scattered data. The default activation in Keras is tanh. In the model that we ended up with as the final version, we adopted the 'relu' activation instead of tanh or sigmoid. ReLU function is now more widely used in the machine learning industry. This is because ReLU is especially advantageous when dealing with gradient vanishing, as the constant gradient would result in faster and more efficient deep neural networks training. Another benefit of ReLu is its sparsity, where several weights could be set to zero. In practice, actually, networks with Relu tend to show better convergence performance than sigmoid [9].

Then we added another layer of LSTM with 64 neurons, with activation ReLu function. Then we continued to stack another four layers, arbitrary picking the number of neurons. As will be described in more details in the "results" section, a stacking like this achieved a pretty good result.

Finally, we compiled the layers using the loss function "mean squared errors'," and Adam as the optimizer.

Besides the baseline codes posted in the Google Drive, we also read and learned from the tutorial by Srivastava [1] and Thomas [2].

4.2.2 Prepare and reshape data

We devoted a considerable amount of time into preparing and reshaping data, as a LSTM network expects the input to be in the form [samples, time steps, features], at least in the Keras library. According to the documentation, samples is the number of data points, time steps is the number of time-dependent steps that are there in a single data point, and features refers to the number of variables for the corresponding true value in training output.

In the beginning, we took the training input, mapped all alphabets from the sequence to number representations. Then we concatenated two sequences into one for simplicity. LSTM model required all of the inputs have the same length, so we transformed all of the inputs to the maximum column size by adding zero. This method may lead to noises since we added so many zeros; however, we had to pad the train data to get it into the LSTM model.

For training output, we took the average value of each matrix, namely an array of 1382 values.

The test input was transformed in the same way as the training input.

4.3. Results

In the final result, as we output average values for the matrices, we simply refilled the number back to construct the matrix, adding zeros on the diagonal. An example result is presented in Figure 3. The RMSE implied okay result, however, we question if simply predicting the averages would be meaningful, as it captures no information with regard to the variations in a single matrix. Thus, we thought about the ways through which our method could be improved, which was discussed in detail in the following two sections, the "trail and errors," and "potential improvement" section.

For understanding how our model performed, we set up the validation split as 0.2, and epoch as 15 when we fitted the model. As you can see, Figure Four shows the curve for the loss value and the validation loss value. The value drops drastically after a few epochs and two lines converges. We believe LSTM turns out working well.

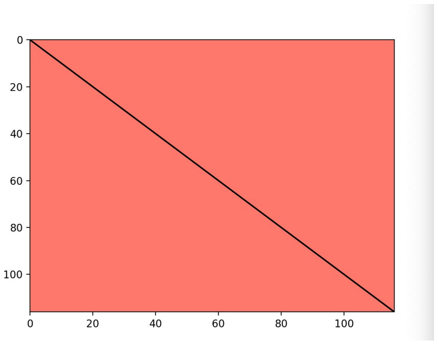

Figure 3. Example Output Matrix from the LSTM model

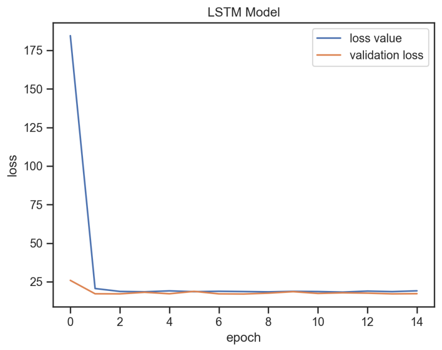

Figure 4. Loss curves from the LSTM model

4.4. Trail and errors

Here we recorded out attempts and failures with LSTM modeling.

After successfully implemented the LSTM model for predicting averages, we wanted to improve our LSTM model with extracting the downward (or upward) triangle from the training output as the output for the LSTM model. Since the last version of LSTM model, we used the average length of the amino acid as the training output for LSTM, we thought extracting part of the matrix can help us get detailed training output that we can later construct a complete matrix out of it. We also tried to use the whole matrix as the output for LSTM. We specifically focus on the trial of the triangle since the matrix is symmetric by the diagonal, we could train the LSTM to get information from the triangle. By applying the LSTM with two LSTM and three dense layers, we used the triangle and the whole matrix as our outputs. For these two trials, we got the same error messages saying that we set an array element with a sequence. We speculated the reason behind the error. LSTM did not accept the shape of our output array/matrix. If we wanted to train the sequence to get the matrix, we could decompose the matrix and train arrays within the matrix and trained them multiple times. Unfortunately, our ideas were severely limited by our ability in Python coding, especially data/matrix manipulation.

Our team member also came up with another attempt to break down the number of examples in each trial like selecting the first (index = 0) sequence and the first output triangle. Surprisingly, when we fitted the model, it worked and did not print out the error message. However, we did not come up with a thorough methodology for training sequence one by one since it would become a problem if we have 4554 models. As time was restraint, we didn't go further down into this logic.

4.5. Potential improvement

One important improvement for our current version of LSTM modeling is to figure out how to output data that is not simply an average value. As described in the above session, we did not end up getting it through.

Another improvement we could think of is to increase the complexity in the LSTM model itself, if given more time.

5. Automated Machine Learning

In addition to KNN and LSTM, at some time point we resorted to automated machine learning. Automated machine learning (AutoML), by having the computer to automatically find the optimal or near-optimal model and parameters for the problem, offers the advantages of producing simpler solutions, faster creation of those solutions, and models that often outperform models that were designed by hand. It would be interesting to see if the results from AutoML would beat the best method designed and constructed purposefully in the competition. Auto-ml is definitely not designed to replace the hard work by real-life data scientists, but it is still meaningful as it would free the data scientists from selecting models to work on sometimes more essential aspects of problem, such as acquiring data and interpreting the model results. However, we note that Machine learning is only one part of the data science process, and it still takes a human brain to put all the different aspects of a problem together in the end.

We spent a considerable time trying to install, implement, and figure out two automated machine learning libraries, auto-sklearn, an extension of the Python library scikit-learn which is a drop-in replacement for regular scikit-learn classifiers and regressors, and TPOT, a "data-science assistant" which optimizes machine learning pipelines using genetic programming.

5.1. Auto-sklearn

Auto-sklearn frees a machine learning user from algorithm selection and hyperparameter tuning. It leverages recent advantages in Bayesian optimization, meta-learning and ensemble construction [7]. We pretty much hoped for implementing it successfully and get output through it regression functions to approximate the best averages for the matrices.

However, installing auto-sklearn library impeded our way. We have tried so many ways, but we ended up failing installing the library. Since all of us used Mac machine and did not have the Linux system installed, we encountered all kinds of errors while installing. We also tried to use the windows machines in the Bobst Library, but it did not work on those machines, either. We even tried to download a simulated application for the Linux system. However, none of them work. We speculated the core problem was that the auto-sklearn installation required linux distribution and C++ compliers, which we were unable to tackle.

5.2. TPOT

Therefore, we decided to try another auto machine learning library called TPOT (Tree-based Pipeline Optimization Tool). It could help us with both classification and regression problems. We selected regression as our method by using TPOTRegressor. The TPOTRegressor performed an automatic search over machine learning pipelines that could contain supervised regression models, feature selection techniques, and any other estimator or transformer that followed the scikit-learn API. The TPOTRegressor would also the hyperparameters of all objects in the pipeline.[4]

Even though TPOPRegressor could automate most of the processes itself, we still needed to customize the automation. Firstly, we set the generations to be 100, which was the number of iterations to the run pipeline optimization process. Generally, TPOT would work better when we gave it more generations (and therefore time) to optimize the pipeline. Next, we set the mean squared error as the function used to evaluate the quality of a given pipeline for the regression problem. Besides, we set the maximum minute. TPOT optimized the pipeline to 480 minutes and limited the maximum minute, and it evaluated a single pipeline to 5 minutes. Setting this parameter to higher values would allow TPOT to evaluate more complex pipelines, but would also help TPOT to run longer. The codes successfully worked, but we still did not get the final result because the optimization process stopped.

5.3. Other AutoML tools

Besides, we signed up for the Google Cloud Platform for the automatic cloud engine developed by Google. The engine could run Sci-kit learn and XGBoost automatically and find the best model. However, since the service was not free and required a complicated installation, we may consider it as a potential plan to improve our model once we have more time to try the Google auto machine learning platform.

6. Summary

In summary, we tried three machine learning methods for the question in hand, which was to predict distance matrices for protein structure. We were given a training set of 4,454 examples with labels of distance matrices. Our baseline method, K Nearest Neighbors, met our most basic expectations, although the RMSE was quite large. Also trying LSTM and automated machine learning, out best result was achieved by constructing multi-layered LSTM.

References

[1] Essentials of deep learning : Introduction to long short term memory. `https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/`. Accessed: 2018-11-29.

[2] Keras lstm tutorial how to easily build a powerful deep learning language model. `http://adventuresinmachinelearning.com/keras-lstm-tutorial/`. Accessed: 2018-11-29.

[3] Longest common substring algorithm. `https://www.bogotobogo.com/python/python_longest_common_substring_lcs_algorithm_generalized_suffix_tree.php`. Accessed: 2018-11-07.

[4] Tpot documentation. `https://epistasislab.github.io/tpot/api/#regression`. Accessed: 2018-11-28.

[5] Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2018-11-22.

[6] R. Casey. Blast sequences aid in genomics and proteomics. *Business Intelligence Network*, 2005.

[7] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

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

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.