Challenges in Bioinformatics

Classification of Non-Ribosomal Peptide Syntheses with Feed-Forward Neural Networks

Dan Ariel Søndergaard Torben Muldvang Andersen
Henrik Schmidt-Møller

Friday 25th October, 2013

Abstract

Non-ribosomal peptide synthetases (NRPSs) are enzymes that primarily occur in bacterial and fungi organisms. These enzymes produce a number of non-ribosomal peptides (NRPs) which have quite diverse characteristics. It has been found that the A-domain of a NRPS module is crucial to the end result of the process. Which product is produced by a NRPS can be concluded experimentally, but this is a slow and expensive process, which makes prediction of the NRPS product quite important.

In this report we describe the basic theory of neural networks for regression and classification, how to train such neural networks, and then apply a neural network for classification of NRPS sequences. To be able to input sequences of varying length into the neural network we have constructed an encoding mapping any sequence to a fixed size vector by counting the occurences of all possible n-grams in the sequence.

Our implementation, nNRPS, is built using PyBrain, a modular machine learning library for the Python programming language.

We have evaluated nNRPS using five-fold cross validation for n-grams of different sizes and found out that 3-grams gives us the best result. Further we have compared our result to two other classifiers based on profile hidden Markov models. We obtain an accuracy of 80.5% whereas the two other classifiers have 86.0% and 86.4% true predictions.

Contents

1 Introduction

Non-ribosomal peptides (NRPs) are products synthesized by a non-ribosomal process that mainly occurs in bacterial and fungi organisms. The diversity of these products is quite large and thus NRPs are of significant interest for e.g. the pharmaceutical industry in the production of antibiotics.

A non-ribosomal peptide synthetase (NRPS) is a large enzyme which consists of a number of modules. The minimal module consists of three domains: the A-domain, the PCP-domain, and the C-domain. The A-domain is the most important in the scope of this report, as it is responsible for recruiting the amino acids that are incorporated into the final product. Thus, the order of A-domains in the assembly line of a NRPS determines the sequence of amino acids in the produced NRP (Eddy, 1998).

This project is based on theq data set presented by Prieto et al. (2012), which consists of 1546 sequences corresponding to A-domains and their product. We use this mapping to build a classifier which uses a feed-forward neural network to predict the product of a certain A-domain.

An artificial neural network is a machine learning technique inspired by the way neurons in the brain interact. Mathematically this corresponds to a form of generalized linear regression, for which efficient optimization algorithms exist.

This report considers the implementation of our classifier, nNRPS, including the theory behind neural networks, methods for their training, and how to use a neural network for classification of sequences. Finally, we evaluate our approach using the k-fold cross-validation method and compare the performance of our approach to existing solutions, particularly the one presented by Prieto et al. (2012).

1.1 Acknowledgements

We wish to thank Thomas Mailund and Christian Nørgaard Storm Pedersen for insightful discussions during this project.

2 Theory

In this chapter we will describe the theory behind neural networks. We will cover feed-forward neural networks and methods for their training, and how a neural network is adapted to the problems of prediction and classification. A general overview of the classification process can be seen in figure 1.

We shall begin by describing linear models of regression since they provide a useful intuition for the mathematics behind neural networks.

2.1 Linear Models of Regression

Suppose we are given set of input variables $\mathbf{X} = \{\mathbf{x}_n\}$ for $n = 1, \ldots, N$ and a corresponding set of target variables $\mathbf{T} = \{\mathbf{t}_n\}$. The goal of regression is then to predict the values of the target variables given a $\mathbf{x}_n$ of dimension D. That is, we wish to construct a function $\mathbf{y} = f(\mathbf{x})$ which given a training set as described above, can predict the target value of a new point $\mathbf{x}'$.

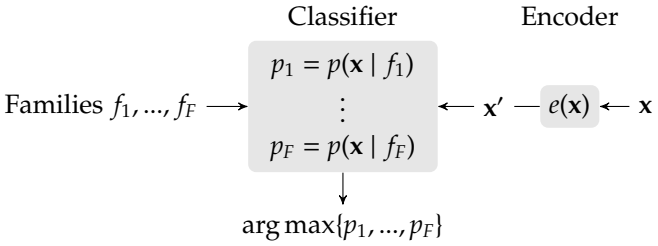

Figure 1: An illustration of the classification process. In this report we present a classifier based on neural networks.

The simplest linear model of regression can be formulated as,

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \cdots + w_D x_D \ .$$

This is also what is usually known as *linear regression*. While this model is useful for some simple cases of data sets, notice that y is not only linear in $\mathbf{w}$, but also in our input vector $\mathbf{x}$. To avoid this limitation in our linear model we may introduce the notion of *basis functions* which we shall denote ϕ_j, a set of fixed non-linear functions on the input values, as follows:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) \ .$$

Notice that the parameter w_0 is special. It allows for a specific offset in the values of y and is called the *bias parameter*. To see why the bias parameter is useful, consider the case where $M = 2$ and ϕ_j is the identity function, and we wish to approximate the linear function $y(x) = 1 + 2x$. In that case our model of regression will be $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 \phi_1(\mathbf{x})$. Without the bias parameter we would never be able to approximate our function exactly, no matter what we choose as $\mathbf{w}_1$.

As we shall see later this concept is also used in neural networks. For convenience we may define $\phi_0(\mathbf{x}) = 1$ to obtain the more readable version,

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\intercal \boldsymbol{\phi}(\mathbf{x}) \ .$$

This is our generalized linear model of regression, and we shall later describe how to expand this model to a feed-forward neural network for regression.

While our model now allows our function $y(\mathbf{x}, \mathbf{w})$ to be a non-linear function of the input vector $\mathbf{x}$, we now have the challenge of choosing a suitable $\boldsymbol{\phi}$. Neural networks accomplish this problem by estimating the basis functions. We will look into how this is done in the next section.

2.2 Neural Networks

A neural network consists of two basic components: neurons and signals. These are then composed to form a complex network of interacting entities. A neuron is an isolated *unit*, which fires a signal when a certain threshold is reached. The

input and output of a neuron is a signal of some strength, modelled by a *weight*. These concepts, units and weights, form the basis of a neural network.

In this section we will describe feed-forward neural networks. These are special neural networks with some constrains on the layout of the network, that is how the units are arranged and which weights are present.

As discussed in the previous section linear models for regression can be expressed as

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x}) \, . \tag{1}$$

This may be extended to classification by evaluating a function on the result of the above equation. Hence, equation 1 may be rewritten as

$$y(\mathbf{x}, \mathbf{w}) = f\left(\mathbf{w}^\top \phi(\mathbf{x})\right) \tag{2}$$

where f is a sigmoidal *activation function* (discussed in section 2.2.3) in case of classification and simply the identity function when doing regression.

As described in Bishop (2006, p. 225–228) neural networks generalize linear models for regression and classification such that we do not need to know the basis functions ϕ. Instead, we can make them depend on some parameters that are adjusted like the weights $\mathbf{w}$ using training. In this way the neural network itself approximates the basis functions exactly like it approximates the weights. A neural network is constructed as follows:

First we construct M linear combinations of the input $x_1, x_2, \ldots, x_D$ as

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{3}$$

for $j = 1, \ldots, M$. Note that we have reintroduced the biases. We will show how to remove those for neural networks later. The superscript (1) indicates that the weights correspond to the first "layer" of the network. $a_1, \ldots, a_M$ are called *activations* and these are passed to a activation function h, so we obtain

$$z_j = h(a_j) \, .$$

$z_1, \ldots, z_M$ are called the *hidden units* of the neural network. These values are used are then used as input to the second layer of the network. So, using equation 1 we obtain

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{4}$$

for $k = 1, 2, \ldots, K$ where K is the number of outputs.

Again we apply an activation function σ on the above values. As for linear models this can be either the identity function if we are doing regression or a sigmoid function when dealing with classification. Thus, for regression we set the output $y_k = a_k$ and for classification $y_k = \sigma(a_k)$.

We can now combine the above steps and obtain the following function

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \, . \tag{5}$$

As it can be seen this looks similar to equation 2. We have simply added an extra layer of computation that estimates the basis functions. Thus, a neural network is a nonlinear function mapping from a set of input values $x_1, \ldots, x_D$ to a set of output values $y_1, \ldots, y_K$, controlled by a set of adjustable weights $\mathbf{w}$. Next, we will absorb the weights into the model like we did in equation 2 to make the formula a bit simpler.

2.2.1 Absorbing the Bias Parameters Into the Model

To simplify the model above we may absorb the weights into the model such that we do not need to explicitly take care of them. This is similar to what we did in 2.1. If we create a new input unit $x_0 = 1$ equation 3 may be written as

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \ .$$

Likewise, if we create another hidden unit $z_0 = 1$ equation 4 becomes

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \ .$$

Thus, using the two expressions above and using matrix multiplication instead of summation, the network function from equation 5 can be written as follows:

$$y(\mathbf{x}, \mathbf{w}) = \sigma\left(\mathbf{w}^{(2)\intercal} h\left(\mathbf{w}^{(1)\intercal} \mathbf{x}\right)\right)$$

which look quite similar to equation 2 (Bishop, 2006, p. 229).

2.2.2 Graphical Representation

Using the notion of units and weights we can represent equation 5 graphically as shown in figure 2. The units on the left correspond to the input values that are then propagated forward to the hidden layer, and the hidden units are propagated to the output units. In this example we have a single hidden layer. In general we may have any number of hidden layers each with an arbitrary number of hidden units.

2.2.3 Choice of Activation Functions

The activation functions are a continuous non linear functions. The continuity of the function ensures that it is differentiable with respect to the parameters of the model. This is crucial when training the network as we shall see in section 2.3.1.

The functions are chosen to be non linear, since a number of linear transformations can be turned into a single linear transformation. Hence, if we use linear activation functions we may as well remove every hidden layer.

Furthermore, the activation functions are chosen to be sigmoidal functions. These are simply functions that are S-shaped. Typically the activation functions

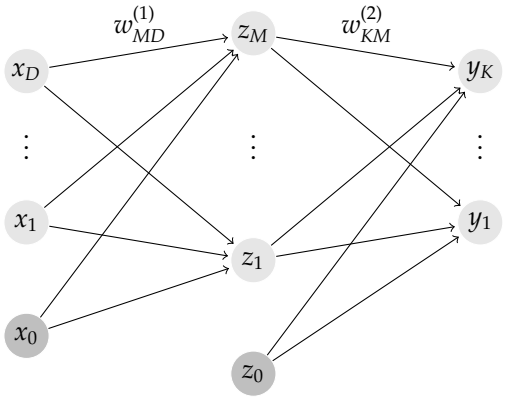

Figure 2: A feed-forward neural network. As input we take a D dimensional vector $\mathbf{x}$ corresponding to $x_1, \ldots, x_D$. x_0 is always set to 1 such that the bias parameters consists of the weights $w^{(1)}_{j0}$. Likewise we have M hidden units and an exstra unit z_0 creating the biasas $w^{(2)}_{k0}$.

of the hidden layers are chosen to be a sigmoid function such as the logistic sigmoid defined as

$$h(a) = \frac{1}{1 + \exp(-a)} .$$

For binary classification problems the activation function σ of the output units are also chosen to be the logistic sigmoid function whereas for multi class problems we use the function

$$\sigma(a_k) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

which is known as the soft-max function (Bishop, 2006, p. 198, 228–229).

2.2.4 Capabilities of Neural Networks

Finally, we may question to approximation capabilities of neural networks. Informally, the universal approximation theorem states that a feed-forward neural network of one hidden layer with a fixed number of hidden units can approximate an arbitrary continuous function. This theorem was first shown by Cybenko (1989) with a few assumptions on the activation functions. Later it was shown by Hornik (1991) that the result is independent of these assumptions.

2.3 Network Training

Training a neural network is the process of adjusting the weights such that a given error function is minimized. One way to do this is to first feed data through the network, calculate the error function on the output, and then adjust the weights "backwards" in the network using some measure derived from the error. This method is called *back propagation*.

In this section we shall see how the back propagation process can be derived for feed-forward neural networks and how the weights can then be adjusted using the obtained gradient. Finally, we present another approach to adjusting the weights, known as *resilient back propagation*, which in many practical use cases can reduce the time to convergence and improve training.

2.3.1 Back Propagation

Back propagation is a method for feed-forward neural networks, to evaluate the gradient of the error, such that the weights of the network can be adjusted accordingly. To obtain the gradient we must derive the error of the error function. This can be done separately for each input pattern. That is, we wish to iteratively minimize $E_n(\mathbf{w})$ for pattern n (Bishop, 2006, p. 241–244).

First, let us consider the error of a single unit. Since E_n depends on the weight w_{ji} only via the summed input a_j to unit j we can use the chain rule on the derivative of E_n with respect to a weight w_{ji},

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \ .$$

Let $\delta_j \equiv \partial E_n / \partial a_j$ where δ_j can be thought of as the error of unit j. Recall that,

$$a_j = \sum_i w_{ji} z_i \ .$$

Differentiating a_j with respect to w_{ji} we obtain,

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \ .$$

The derivative of E_n with respect to w_{ji} is then given by,

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \ .$$

This means that we can evaluate the derivative by using the δ at the output end of the weight multiplied by the z at the input end of the weight, where the weight is represented by the edge w_{ji}. We have now obtained a way of evaluating the gradient. Since z_i is given by forward propagation, we only have to compute δ for all hidden and output units in the network.

Let us first consider the error of the output units in the neural network. For linear output units this is given by,

$$\delta_k = y_k - t_k \ .$$

In order to evaluate δ for every hidden unit we need the *back propagation* formula which is given by (Bishop, 2006, p. 244),

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \ ,$$

where we sum over the error of all units to which unit j has an edge. That is, the value of δ for hidden unit j can be obtained by way of the δ in the layer to

the right along with the weights to their respective units, and the activation of hidden unit j. This approach is called *on-line training* since the weights are adjusted for every pattern one at a time.

A variation is to cycle through the entire data set, accumulating the error one the way, and then adjust the weights accordingly after each "epoch". That is,

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}} \, .$$

Techniques that use this approach are known as *batch* methods.

2.3.2 Gradient Descent

Gradient descent (Bishop, 2006, p. 240) is the process of adjusting the weights of the neural network to reach a minima using the gradient by taking a small step η in the direction of the negative gradient. This is the final step in the training process as we have now evaluated the gradient using back-propagation and are able to adjust the weights using the gradient information. That is,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n \left(\mathbf{w}^{(\tau)} \right)$$

where $\eta > 0$ is a fixed number known as the *learning rate* used through training. Using a fixed learning rate in this way turns out to be problematic. If the learning rate is set too high, one can potentially continuously "overshoot" the local minimum resulting in poor convergence. On the other hand, if the learning rate is set very low convergence can take a long time.

We shall now describe an alternative method of adjusting the weights which improves the stability and convergence-time of the training process.

2.3.3 Resilient Back Propagation

As mentioned earlier using of a global learning rate has a number of drawbacks. If set too high it may cause the algorithm to oscillate around a local minimum. If set too low, convergence may take a long time. In *resilient back propagation* described by Igel and Hüsken (2003) and Riedmiller (1994), the learning rate is replaced with an adjustable step size $\Delta_{ij}^{(\tau)}$ for each unit. In each time step the weights are then adjusted by,

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} + \Delta_{ij}^{(\tau)} \, .$$

The step size Δ_{ij} is adjusted by

$$\Delta_{ij}^{(\tau)} = \begin{cases} \min(\eta^+ \Delta_{ij}^{(\tau-1)}, \Delta_{max}) & \text{if } \frac{\partial E^{(\tau-1)}}{w_{ij}} \frac{\partial E^{(\tau)}}{w_{ij}} > 0 \\ \max(\eta^- \Delta_{ij}^{(\tau-1)}, \Delta_{min}) & \text{if } \frac{\partial E^{(\tau-1)}}{w_{ij}} \frac{\partial E^{(\tau)}}{w_{ij}} < 0 \\ \Delta_{ij}^{\tau-1} & \text{otherwise} \end{cases}$$

where $0 < \eta^- < 1 < \eta^+$ denotes predefined constant step factors. In other words, if the current partial derivative does not change sign the step size is increased by some factor up to a given maximum. If the partial derivative changes sign,

the local minimum has been stepped over so the step size is decreased. $\Delta w_{ij}^{(\tau)}$ is then calculated as,

$$\Delta w_{ij}^{(\tau)} = \begin{cases} -\operatorname{sign}\left(\frac{\partial E^{(\tau)}}{\partial w_{ij}}\right)\Delta_{ij}^{(\tau)} & \text{if } \frac{\partial E^{(\tau-1)}}{w_{ij}}\frac{\partial E^{(\tau)}}{w_{ij}} \geq 0 \\ -\Delta w_{ij}^{(\tau-1)} & \text{if } \frac{\partial E^{(\tau-1)}}{w_{ij}}\frac{\partial E^{(\tau)}}{w_{ij}} < 0 \end{cases} \tag{6}$$

where the sign function returns +1 i its argument is positive, −1 if its argument is negative, and 0 otherwise.

In addition, the error derivative $\frac{\partial E^{(\tau)}}{\partial w_{ij}}$ is set to 0 if $\frac{\partial E^{(\tau-1)}}{w_{ij}}\frac{\partial E^{(\tau)}}{w_{ij}} < 0$ which avoids a step size update on the next iteration. This procedure along with one in 6 is known collectively as *weight backtracking*, because it effectively reverts the weight update if the gradient changes sign.

It can be argued (Igel and Hüsken, 2003) that the additional weight backtracking heuristic is somewhat arbitrary because even though the change in sign of the gradient implies that a local minimum has been stepped over, the backtracking does not take into account whether or not the error has decreased as a result of the weight update. A popular variant known as Rprop- has been proposed which always chooses the first case of 6 regardless of whether the gradient changes sign.

3 Implementation

In this chapter we shall describe the choice of network architecture used in our classifier, nNRPS. The network architecture described has some limitations, most importantly the number of input units in the neural network is fixed at construction time. Thus, to accept an arbitrary input sequence, we must encode all sequences to a vector of fixed dimensionality. We describe our encoding and give a short analysis of its usefulness in this application domain.

Our classifier consists of a single program written in Python 2.7.5 which allows the user to train a neural network, classify sequences given some trained neural network, and perform simple cross-validation. For a detailed description of how to run our programs we refer to chapter C in the appendix.

3.1 Network Architecture

The network architecture used in our classifier is inspired by the architecture presented by Wu et al. (1995), which is a feed-forward neural network with one input layer, three hidden layers of 10 hidden units, and an output layer. All hidden units use the sigmoid hyperbolic tangent activation function. This architecture is successfully applied to protein classification.

For the classification problem it is useful to have one output unit per family, which in our case results in 30 output units. Using the soft-max activation function on the output units means that we may interpret the value of these units as a posterior probability of the sequence belonging to each family. We use Rprop-, as previously described, to adjust the weights.

In section 3.2 we shall see how the number of input units is chosen, since this depends on the chosen encoding. The neural network topology is illustrated in figure 2.

During the cross-validation process we experimented with a variety of configurations. All experiments showed no performance increase if the number of layers or units were increased. In fact it seems that one hidden layer with 10 hidden units resulted in the best performance. Therefore, this is the architecture that we refer to in the rest of the report.

3.2 Encoding of Input

As described in section 1 our data set consists of a number of amino acid sequences. These sequences vary in length and thus we cannot build a single neural network for them all, since the neural network would then need a variable number of input units.

To circumvent this limitation we use an encoding which reduces each sequence to a fixed-length vector. Such an encoding inevitably looses some information about the sequence since it corresponds to dimensionality reduction. A good encoding should throw away as little information as possible and maintain as much of the variability between families as possible.

Our encoding is based on the concept of n-grams, also known as k-mers. We define the n-grams of a sequence to be the set of all sub strings of length n of a string x.

For an alphabet Σ we have $D = |\Sigma|^n$ possible n-grams, so we construct a vector x' of D dimensions such that x'_i is the number of occurrences of the i'th n-gram. This vector is then normalized such that $0 \leq x'_i \leq 1$ by dividing by D. We denote this encoding $\mathbf{x'} = e_n(\mathbf{x})$.

Thus, we able to encode an arbitrary string over Σ to a vector of D dimensions. For our data set we have $|\Sigma| = 20$ and $n = 2$ (see chapter 4) which yields $D = 20^2 = 400$ possible n-grams. The network architecture described above is then extended such that we have 400 input units.

This encoding corresponds to the encoding presented by Wu et al. (1995), but without Singular Value Decomposition (SVD) to further reduce the size of the vector. While this might be valid approach to improving performance of the classifier, we did not have time to implement this.

4 Evaluation

In this section we will discuss our evaluation of nNRPS. We use a typical k-fold cross-validation approach and compare it to the performance of other classifiers evaluated on the same data set.

A k-fold cross-validation partitions the data set into k parts such that each family is represented proportionally to its representation in the entire data set. The classifier is then trained using $k - 1$ parts and the last part is then classified. This is done k times. The accuracy is measured by the number of correct classifications. This test is more conservative than a leave-one-out (LOO) test, since the data used for training is significantly smaller. In this project $k = 5$ was used.

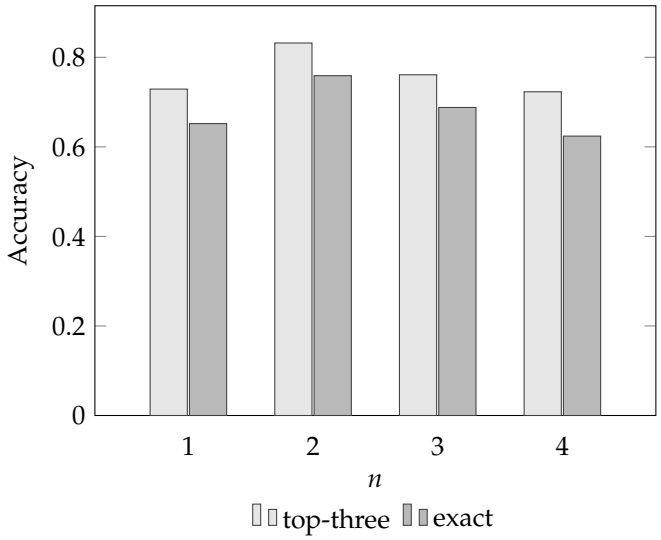

Figure 3: Here we illustrate the accuracy of the classifier for 1, 2, 3, and 4-grams. Observe that considering the three best predictions does not yield a high increase in accuracy, meaning that predictions are either right or very wrong.

4.1 Performance

As previously described we encode the sequences using n-grams. We have evaluated the performance of the classifier using values of 1, 2, 3, and 4-grams. For 5-grams the network became too big to be computable on a normal computer. In the tests we have made 150 iterations for training.

For each n-gram size we have tested if the prediction was "exactly" correct, that is if the output node with the largest value corresponds to the actual family. Further we have tested if the actual family is in the top three largest valued output nodes to see if we "almost" classify the sequences correctly. The results can be seen in figure 3 and table 1 on page 16.

As seen larger n-grams does not make the prediction better. The best result is obtained using 2-grams. If we look at the top three, the accuracies grow by about ten percentage points, so looking at the top three of course makes the predictions better, but there are still many sequences that get classified as a family not in top three.

To see how the neural network performs as a function of the number of iterations used during training, we have performed 5-fold cross-validation for $i = \{16, 32, 64, 128, 256\}$ iterations. To validate that the results were consistent the experiment was repeated five times. The results of this experiment is shown in table 2 and visualized in figure 4 for 2 and 3-grams.

For $n = 2$ the plot clearly illustrates how the neural network training reaches a point of over-training after 128 iterations. For $n = 3$ this event occurs earlier in the training process after 16 iterations.

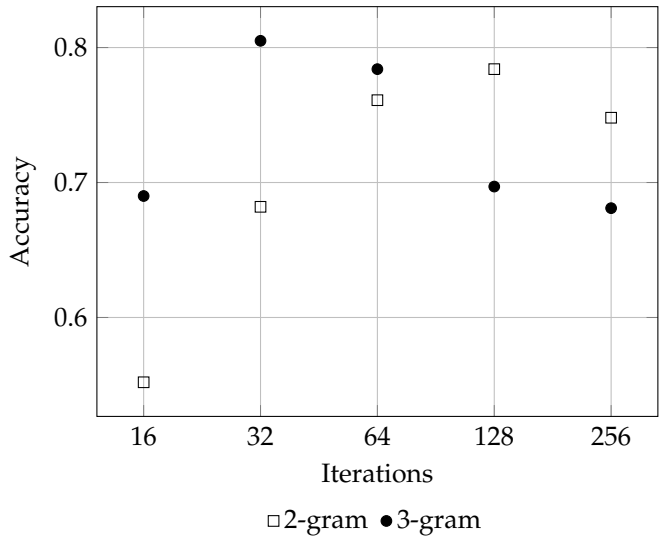

Figure 4: This figures illustrates the performance of the classifier for 2 and 3-grams as a function of the number of iterations. For 2-grams we over-train at 128 iterations, while it happens much earlier for 3-grams at 32 iterations.

4.2 Confusion Matrix

To further analyse the prediction power of our classifier we have constructed a *confusion matrix* (see table 4 on page 19). The rows and columns of the confusion matrix represent the 30 families of the data set such that cell (i, j) contains the number of sequences which true family is i and was classified as j. The diagonal thus contains the number of correctly predicted sequences for each family.

Thus, it is interesting to observe the confusion matrix from two points of view: row and column-wise. Observing row i tells us the distribution of the predictions of sequences in family i. An interesting example is the F and Y families for which predictions are spread over a wide range of families.

This observation corresponds to what can be observed in the phylogenetic tree found in figure 6 which might indicate that a classifier cannot improve much over the current results with the given data set.

If we observe the confusion matrix column-wise we can see which family "swallows" predictions in the classifier. Again, F is an interesting example since most entries in the F-column are non-zero. That is, a lot of sequences are wrongly classified as F which is also consistent with the distribution of sequences of the F family in figure 6.

5 Previous Works

Prieto et al. (2012) presents NRPSsp, a predictor based on profile HMMs. Instead of using k-fold cross-validation, they have used leave-one-out for their analysis. Thus the results should be taken with a grain of salt.

In the article they compare NRPSsp to NRPSpredictor2 (Röttig et al., 2011). The training data set is not the same for those two, since new sequences have been added to the on-line databases, so the predictors are not fully comparable. By performing leave-one-out NRSPsp obtains an accuracy of 86.4%, while NRPSpredictor2 only reaches 77.4% (Prieto et al., 2012).

Søndergaard and Andersen (2013) presents pyNRPS, a predictor also based on profile HMMs. By making a 5-fold cross-validation on exactly the same data set as we have used in this project, pyNRPS obtains an accuracy of 86.0%.

By making a five-fold validation we obtain an accuracy of 80.5%, which is not as good as either NRPSsp or pyNRPS, but we believe that further optimizations may increase the performance of the classifier.

As seen in table 3 at page 17 some of the large families like A and dhb are predicted almost correct, whereas many of the smaller families have an accuracy below 50%. To make an overview over how well we classify when looking at families seperately, we have made the histogram shown in figure 5. This figure shows for each 5% interval, i.e. [0–5], [5–10], . . . , [95–100], how many familes that have an accuracy inside that interval. Ideally all families would have an accuracy of approximately 80%, but as seen it varies at lot. We have made the same experiment for pyNRPS, which also have a large variation of accuracy between families, although it is defenitely better than nNRPS.

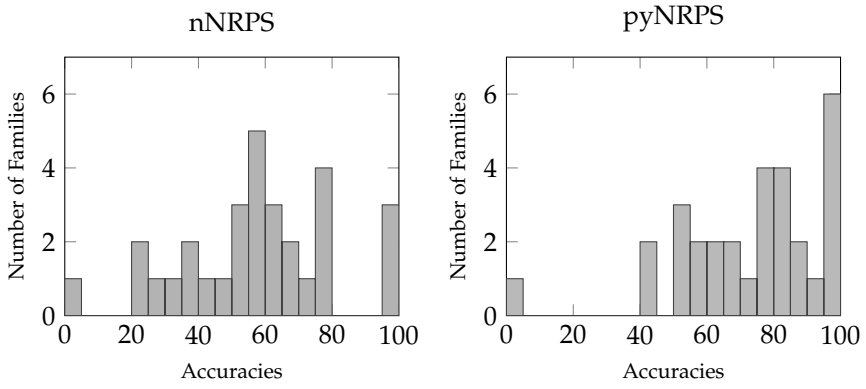

Figure 5: This figure shows for intervals of size 5 between 0 and 100% the number of families with a prediction accuracy inside each interval.

6 Conclusion

In recent years there has been a rapid expansion the amount of data that needs to be processed and analysed in the field of bioinformatics. Part of this immense data flow consists of the NRPS sequence data which we consider in this report and which have recently gained notice from the pharmaceutical industry, among others, because of the diversity of substrates that these NRPSs produce.

In this project we have considered the problem of predicting the product of a given NRPS module, based on the sequence of its A-domain. Because the input sequences can have variable length and a neural network has a fixed number of input units, each sequence is encoded by an approach based on

n-grams of the sequence. A neural network was then trained to predict the product of the encoded sequences. Finally, five-fold cross validation was used to evaluate the result.

Experiments showed that varying the number of hidden units and layers made little difference in the prediction power of the neural network. Thus, they have not been included in this report.

Wu et al. (1995) suggests using Singular Value Decomposition (SVD) to reduce the dimensionality of the input vector. We did not implement this due to time constraints. However, one item of future interest may be to perform SVD on each family. When a new sequence must be encoded, it is then encoded once for each family using the respective SVD decomposition, and each encoded sequence then is propagated through the network simultaneously. This might help capture important features of each family better than our current approach.

The classifier presented here achieves an accuracy of 80.5% in its current state, but is inferior to the Profile HMM approaches presented by Søndergaard and Andersen (2013) and Prieto et al. (2012) which achieve accuracies of approximately 86%. However, we believe that the application of SVD and fine-tuning of the number of iterations used to train the network might increase the performance of our classifier so that it can compete with the Profile HMM approaches. Finally, the phylogeny presented in this report suggests that the data set may be too small and inconsistent to substantially improve the performance of classification based on any machine learning approach.

A Accuracy Results

n-gram	Type	True	False	Accuracy
1-gram	exact	1008	538	0.652
2-gram	exact	1173	373	0.759
3-gram	exact	1063	483	0.688
4-gram	exact	964	582	0.624
1-gram	top-three	1127	419	0.729
2-gram	top-three	1287	259	0.832
3-gram	top-three	1176	370	0.761
4-gram	top-three	1118	428	0.723

Table 1: This table shows the results of 5-fold cross-validation for $n = \{1, 2, 3, 4\}$. With exact we only consider the single most highly rated result, while top-three considers the three most higly rated results. If one of them is the true result, it is counted as a correct prediction.

n-gram	i	True	False	Accuracy
2	16	854	692	0.552
2	32	1055	491	0.682
2	64	1176	370	0.761
2	128	1212	334	0.784
2	256	1156	390	0.748
3	16	1067	479	0.690
3	32	1245	301	0.805
3	64	1212	334	0.784
3	128	1077	469	0.697
3	256	1053	493	0.681

Table 2: This table illustrates the accuracy for various n and number of iterations i. It is slightly surprising that $n = 3$ performs better than $n = 2$ for $i = 32$. This may be due to over-fitting when $i > 32$.

Product	True	False	Accuracy
A	604	14	0.98
C	23	14	0.62
D	16	12	0.57
E	10	22	0.31
F	49	50	0.49
G	16	11	0.59
I	8	7	0.53
K	4	4	0.50
L	30	11	0.73
N	13	9	0.59
P	12	10	0.55
Q	5	5	0.50
R	4	2	0.67
S	19	15	0.56
T	30	9	0.77
V	24	14	0.63
W	6	10	0.38
Y	5	17	0.23
aad	54	16	0.77
beta-ala	0	5	0.00
bht	4	5	0.44
dab	9	3	0.75
dhb	258	8	0.97
dhpg	6	3	0.67
dht	1	3	0.25
horn	3	2	0.60
hpg	21	1	0.95
hyv-d	3	1	0.75
orn	6	11	0.35
pip	2	7	0.22

Table 3: Per family accuracies for a 5-fold cross-validation with 3-grams and 32 iterations.

B Confusion Matrix

	A	C	D	E	F	G	I	K	L	N
A	**604**	0	2	0	5	4	0	0	1	0
C	1	**23**	0	0	8	0	0	0	0	0
D	0	0	**16**	0	10	0	0	0	0	0
E	2	4	1	**10**	9	1	0	0	1	1
F	5	2	7	2	**49**	6	0	1	10	1
G	0	0	2	1	5	**16**	0	0	1	1
I	0	0	0	0	1	0	**8**	2	4	0
K	0	0	0	0	3	0	0	**4**	0	0
L	0	0	1	0	7	0	1	0	**30**	0
N	0	1	0	0	6	1	0	0	1	**13**
P	1	0	2	0	5	0	0	0	1	0
Q	0	0	0	0	3	0	0	0	1	0
R	0	0	0	0	1	1	0	0	0	0
S	0	0	0	0	5	0	0	1	0	0
T	0	1	1	0	3	0	0	0	0	2
V	0	0	0	0	5	1	0	0	5	0
W	0	0	1	0	5	2	0	0	0	0
Y	0	0	1	0	7	0	0	0	3	1
aad	2	0	2	0	5	0	0	0	0	0
beta-ala	1	0	0	0	1	0	1	0	0	0
bht	0	0	0	0	2	0	1	0	0	0
dab	0	0	0	0	2	0	0	0	0	1
dhb	1	0	0	0	5	0	0	0	0	0
dhpg	0	0	2	0	0	0	0	0	0	0
dht	0	0	0	0	0	0	0	0	0	0
horn	0	0	0	0	0	0	0	1	0	0
hpg	0	0	0	0	0	0	1	0	0	0
hyv-d	0	0	0	0	1	0	0	0	0	0
orn	3	0	1	0	6	0	1	0	0	0
pip	0	0	1	0	2	1	0	0	0	0

	P	Q	R	S	T	V	W	Y	aad	beta-ala
A	1	1	0	0	0	0	0	0	0	0
C	0	0	0	0	0	3	0	0	1	0
D	0	0	0	0	1	0	1	0	0	0
E	3	0	0	0	0	0	0	0	0	0
F	1	4	0	2	2	0	3	1	0	0
G	1	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0
K	0	1	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	1	0	0
N	0	0	0	0	0	0	0	0	0	0
P	**12**	0	0	0	0	0	0	0	0	0
Q	0	**5**	0	1	0	0	0	0	0	0
R	0	0	**4**	0	0	0	0	0	0	0
S	0	1	0	**19**	1	0	5	2	0	0
T	0	0	0	0	**30**	0	0	0	0	0
V	1	0	0	0	0	**24**	1	0	0	1
W	0	1	0	0	0	0	**6**	0	0	0
Y	0	0	0	1	0	1	0	**5**	0	0
aad	2	3	0	0	1	0	0	0	**54**	0
beta-ala	0	0	0	1	0	0	0	0	0	**0**
bht	0	0	0	0	0	1	0	1	0	0
dab	0	0	0	0	0	0	0	0	0	0
dhb	0	0	0	0	0	0	0	0	0	0
dhpg	0	0	0	0	0	0	0	0	0	0

dht	0	0	0	0	3	0	0	0	0	0
horn	0	0	0	1	0	0	0	0	0	0
hpg	0	0	0	0	0	0	0	0	0	0
hyv-d	0	0	0	0	0	0	0	0	0	0
orn	0	0	0	0	0	0	0	0	0	0
pip	1	0	0	0	0	1	0	0	0	0

	bht	dab	dhb	dhpg	dht	horn	hpg	hyv-d	orn	pip
A	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	1
D	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	3	0
G	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	1	0	0	0
N	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	1
Q	0	0	0	0	0	0	0	0	0	0
R	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0
T	0	0	0	0	2	0	0	0	0	0
V	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	0	0	1
Y	3	0	0	0	0	0	0	0	0	0
aad	0	0	0	0	0	0	0	0	1	0
beta-ala	0	0	0	0	0	0	0	0	0	1
bht	**4**	0	0	0	0	0	0	0	0	0
dab	0	**9**	0	0	0	0	0	0	0	0
dhb	0	0	**258**	0	0	0	0	0	0	2
dhpg	0	0	0	**6**	0	0	1	0	0	0
dht	0	0	0	0	**1**	0	0	0	0	0
horn	0	0	0	0	0	**3**	0	0	0	0
hpg	0	0	0	0	0	0	**21**	0	0	0
hyv-d	0	0	0	0	0	0	0	**3**	0	0
orn	0	0	0	0	0	0	0	0	**6**	0
pip	0	0	1	0	0	0	0	0	0	**2**

Table 4: Confusion matrix for a 5-fold cross-validation for $n = 3$ and $i = 32$. The diagonal is marked in bold. The confusion matrix shows how many sequences were classified as each family. E.g. we see that sequences of the F family is mainly predicted correctly, and that the wrong predictions are spread over a wide range of other families.

C Using the Classifier

The classifier presented in this report, nNRPS, is written in Python 2.7.5 and requires the PyBrain machine learning library which can be obtained from `http://pybrain.org`. PyBrain can easily be installed using the `easy_install` or `pip` commands which are part of most Python distributions.

The included archive consists of a number of scripts and data files which make up nNRPS. The archive can also be downloaded from `http://users-cs.au.dk/das/nNRPS.zip`. The main script is the `nnrps.py` for which usage information is included below.

```
$ ./nnrps.py -h
usage: nnrps.py [-h] [-v] [-d DIRECTORY] {train,classify,validate} ...

$ ./nnrps.py train -h
usage: nnrps.py train [-h] [-i ITERATIONS] [-o OUTPUT]

$ ./nnrps.py classify -h
usage: nnrps.py classify [-h] [-n NETWORK] [-f FILE]

$ ./nnrps.py validate -h
usage: nnrps.py validate [-h] [-n NGRAM] [-k FOLDS] [-i ITERATIONS] [-t]
```

In addition to the scripts we also include the sequences directory which contains the entire data set.

D Phylogeny

Figure 6: Condensed phylogeny built from the data set. Each sequence was renamed to its family name to make it easier to locate families in the tree. We have recursively merged all leafs which had the same family and added the number of merged leafs to the name. That is, dhb/7.2 means that 7 leafs were merged, and that this is the 2nd occurence of dhb in the tree. Branch lenghts are arbitrary.

Bibliography

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Christian Igel and Michael Hüsken. Empirical evaluation of the improved rprop learning algorithms. *Neurocomputing*, 50:105–123, 2003.

Carlos Prieto, Carlos García-Estrada, Diego Lorenzana, and Juan Francisco Martín. Nrpssp: non-ribosomal peptide synthase substrate predictor. *Bioinformatics*, 28(3):426–7, Feb 2012. doi: 10.1093/bioinformatics/btr659.

Martin Riedmiller. Rprop-description and implementation details. Technical report, 1994.

Marc Röttig, Marnix H Medema, Kai Blin, Tilmann Weber, Christian Rausch, and Oliver Kohlbacher. Nrpspredictor2–a web server for predicting nrps adenylation domain specificity. *Nucleic Acids Res*, 39(Web Server issue): W362–7, Jul 2011. doi: 10.1093/nar/gkr323.

Dan Søndergaard and Torben Muldvang Andersen. Classification of non-ribosomal peptide synthesis substrates with profile hidden markov models. 2013.

Cathy Wu, Michael Berry, Sailaja Shivakumar, and Jerry McLarty. Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition. *Machine Learning*, 21(1-2):177–193, 1995.