Lecture 7 - Neural Networks

Adnan Ferdous Ashrafi

Stamford University Bangladesh

Table of Contents

1 Neural Networks
- Definition
- Inspiration
- Domain of Neural Networks

2 Input and Output Encoding
- Definition
- Continuous Variables
- Categorical Variables
- Output
- Multiple Output nodes

3 Perceptron

4 NEURAL NETWORKS
- ESTIMATION AND PREDICTION
- Example
- Architecture of Neural Networks
- Number of Nodes
  - Hidden Layer
  - Input Layer
- Example

5 Sigmoid Activation Function

6 BACK-PROPAGATION
- Squared Errors

7 GRADIENT DESCENT METHOD
- Working Method

8 BACK-PROPAGATION RULES

9 EXAMPLE OF BACK-PROPAGATION

10 TERMINATION CRITERIA

11 Learning Rate

12 MOMENTUM

13 SENSITIVITY ANALYSIS

14 APPLICATION OF NEURAL NETWORK MOD-ELING

15 Further Reading

16 References

Neural Networks

The inspiration for neural networks was the recognition that complex learning systems in animal brains consisted of closely interconnected sets of neurons. Although a particular neuron may be relatively simple in structure, dense networks of interconnected neurons could perform complex learning tasks such as classification and pattern recognition. The human brain, for example, contains approximately 10^{11} neurons, each connected on average to $10,000$ other neurons, making a total of $1,000,000,000,000,000 = 10^{15}$ synaptic connections.

Definition

Artificial neural networks (hereafter, neural networks) represent an attempt at a very basic level to imitate the type of nonlinear learning that occurs in the networks of neurons found in nature.

Neural Networks- Inspiration

As shown in Figure 1, a real neuron uses dendrites to gather inputs from other neurons and combines the input information, generating a nonlinear response ("firing") when some threshold is reached, which it sends to other neurons using the axon.

Figure 1 also shows an artificial neuron model used in most neural networks.

Neural Networks- Inspiration

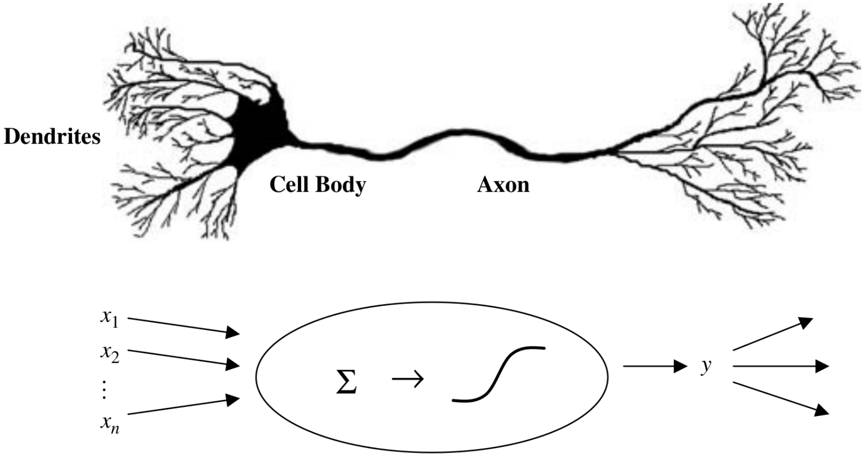


Figure 1: Real neuron and artificial neuron model. (Sketch of neuron courtesy of Chantal Larose.)

Neural Networks- Inspiration

The inputs (x_i) are collected from upstream neurons (or the data set) and combined through a combination function such as summation $(\sum)$, which is then input into a (usually nonlinear) activation function to produce an output response (y), which is then channeled downstream to other neurons.

Neural Networks- Domain of Neural Networks

What types of problems are appropriate for neural networks?

One of the advantages of using neural networks is that they are quite robust with respect to noisy data. Because the network contains many nodes (artificial neurons), with weights assigned to each connection, the network can learn to work around these uninformative (or even erroneous) examples in the data set.

However, unlike decision trees, which produce intuitive rules that are understandable to nonspecialists, neural networks are relatively opaque to human interpretation, as we shall see. Also, neural networks usually require longer training times than decision trees, often extending into several hours.

Input and Output Encoding- Definition

One possible drawback of neural networks is that all attribute values must be encoded in a standardized manner, taking values between zero and 1, even for categorical variables. Later, when we examine the details of the back-propagation algorithm, we shall understand why this is necessary.

For now, however, how does one go about standardizing all the attribute values?

Input and Output Encoding- Continuous Variables

For continuous variables, this is not a problem, as we discussed in Lecture 2. We may simply apply the *min–max normalization*:

$$X* = \frac{X - \min(X)}{\text{range}(X)} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

This works well as long as the minimum and maximum values are known and all potential new data are bounded between them. Neural networks are somewhat robust to minor violations of these boundaries. If more serious violations are expected, certain ad hoc solutions may be adopted, such as rejecting values that are outside the boundaries, or assigning such values to either the minimum or maximum value.

Input and Output Encoding- Categorical Variables

Categorical variables are more problematical, as might be expected. If the number of possible categories is not too large, one may use indicator (flag) variables.

For example, many data sets contain a $gender$ attribute, containing values $female$, $male$, and $unknown$. Since the neural network could not handle these attribute values in their present form, we could, instead, create $indicator$ variables for female and male. Each record would contain values for each of these two $indicator$ variables. Records for females would have a value of 1 for $female$ and 0 for $male$, while records for males would have a value of 0 for $female$ and 1 for $male$. Records for persons of unknown gender would have values of 0 for $female$ and 0 for $male$.

In general, categorical variables with k classes may be translated into $k-1$ indicator variables, as long as the definition of the indicators is clearly defined.

Input and Output Encoding- Categorical Variables

Be wary of re-coding un-ordered categorical variables into a single variable with a range between zero and 1.

For example, suppose that the data set contains information on a marital status attribute. Suppose that we code the attribute values divorced, married, separated, single, widowed, and unknown, as 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0, respectively. Then this coding implies, for example, that divorced is "closer" to married than it is to separated, and so on.

The neural network would be aware only of the numerical values in the marital status field, not of their pre-encoded meanings, and would thus be naive of their true meaning. Spurious and meaningless findings may result.

Input and Output Encoding- Output

With respect to output, we shall see that neural network output nodes always return a continuous value between zero and 1 as output.

How can we use such continuous output for classification?

Many classification problems have a dichotomous result, an up-or-down decision, with only two possible outcomes. For example, “Is this customer about to leave our company’s service?” For dichotomous classification problems, one option is to use a single output node (such as in Figure 3), with a threshold value set a priori which would separate the classes, such as “leave” or “stay.” For example, with the threshold of “leave if output ≥ 0.67,” an output of 0.72 from the output node would classify that record as likely to leave the company’s service.

Input and Output Encoding- Output

Single output nodes may also be used when the classes are clearly ordered. For example, suppose that we would like to classify elementary school reading prowess based on a certain set of student attributes. Then we may be able to define the thresholds as follows:

- If $0 \leq$ output < 0.25, classify *first-grade* reading level.
- If $0.25 \leq$ output < 0.50, classify *second-grade* reading level.
- If $0.50 \leq$ output < 0.75, classify *third-grade* reading level.
- If output > 0.75, classify *fourth-grade* reading level.

Fine-tuning of the thresholds may be required, tempered by experience and the judgment of domain experts.

Input and Output Encoding- Multiple Output nodes

Not all classification problems, however, are soluble using a single output node only. For instance, suppose that we have several unordered categories in our target variable, as, for example, with the marital status variable above. In this case we would choose to adopt 1-of-n output encoding, where one output node is used for each possible category of the target variable. For example, if marital status was our target variable, the network would have six output nodes in the output layer, one for each of the six classes divorced, married, separated, single, widowed, and unknown. The output node with the highest value is then chosen as the classification for that particular record.

Input and Output Encoding- Multiple Output nodes

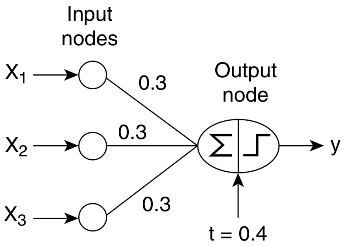
One benefit of using 1-of-n output encoding is that it provides a measure of confidence in the classification, in the form of the difference between the highest-value output node and the second-highest-value output node. Classifications with low confidence (small difference in node output values) can be flagged for further clarification.

Perceptron

Consider the diagram shown in Figure 2. The table on the left shows a data set containing three boolean variables (x_1, x_2, x_3) and an output variable, y, that takes on the value -1 if at least two of the three inputs are zero, and +1 if at least two of the inputs are greater than zero.

X_1	X_2	X_3	y
1	0	0	−1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	−1
0	1	0	−1
0	1	1	1
0	0	0	−1

(a) Data set



(b) Perceptron

Figure 2: Modeling a boolean function using a perceptron

Perceptron

Figure 2(b) illustrates a simple neural network architecture known as a *perceptron*.

The perceptron consists of two types of nodes: input nodes, which are used to represent the input attributes, and an output node, which is used to represent the model output. The nodes in a neural network architecture are commonly known as neurons or units. In a perceptron, each input node is connected via a weighted link to the output node. The weighted link is used to emulate the strength of synaptic connection between neurons. As in biological neural systems, training a perceptron model amounts to adapting the weights of the links until they fit the input-output relationships of the underlying data.

Perceptron

A perceptron computes its output value, $\hat{y}$, by performing a weighted sum on its inputs, subtracting a bias factor t from the sum, and then examining the sign of the result. The model shown in Figure 2(b) has three input nodes, each of which has an identical weight of 0.3 to the output node and a bias factor of $t = 0.4$. The output computed by the model is

$$\hat{y} = \begin{cases} 1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 > 0; \\ -1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 < 0 \end{cases}$$

For example, if $x_1 = 1, x_2 = 1, x_3 = 0$, then $\hat{y} = +1$ because $0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4$ is positive. On the other hand, if $x_1 = 0, x_2 = 1, x_3 = 0$, then $\hat{y} = -1$ because the weighted sum subtracted by the bias factor is negative.

NEURAL NETWORKS- ESTIMATION AND PREDICTION

Clearly, since neural networks produce continuous output, they may quite naturally be used for estimation and prediction.

Suppose, for example, that we are interested in predicting the price of a particular stock three months in the future. Presumably, we would have encoded price information using the min–max normalization above. However, the neural network would output a value between zero and 1, which (one would hope) does not represent the predicted price of the stock.

NEURAL NETWORKS- ESTIMATION AND PREDICTION

Rather, the min–max normalization needs to be inverted, so that the neural network output can be understood on the scale of the stock prices. In general, this denormalization is as follows:

$$\text{prediction} = \text{output}(\text{data range}) + \text{minimum}$$

where output represents the neural network output in the (0,1) range, data range represents the range of the original attribute values on the non-normalized scale, and minimum represents the smallest attribute value on the non-normalized scale. For example, suppose that the stock prices ranged from \$20 to \$30 and that the network output was 0.69. Then the predicted stock price in three months is

$$\text{prediction} = \text{output}(\text{data range}) + \text{minimum} = 0.69(\$10) + \$20 = \$26.90$$

NEURAL NETWORKS- Example

Let us examine the simple neural network shown in Figure 3. A neural network consists of a *layered*, *feed-forward*, *completely connected* network of artificial neurons, or *nodes*.

The feed-forward nature of the network restricts the network to a single direction of flow and does not allow looping or cycling. The neural network is composed of two or more layers, although most networks consist of three layers: an *input* layer, a *hidden* layer, and an *output* layer. There may be more than one hidden layer, although most networks contain only one, which is sufficient for most purposes.

The neural network is completely connected, meaning that every node in a given layer is connected to every node in the next layer, although not to other nodes in the same layer. Each connection between nodes has a weight (e.g., W_{1A}) associated with it. At initialization, these weights are randomly assigned to values between zero and 1.

NEURAL NETWORKS- Architecture of Neural Networks

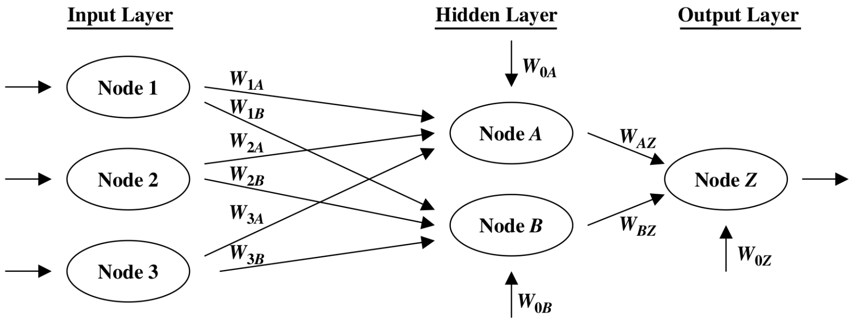


Figure 3: Simple neural network.

NEURAL NETWORKS- Number of Nodes

The number of input nodes usually depends on the number and type of attributes in the data set.

The number of hidden layers, and the number of nodes in each hidden layer, are both configurable by the user.

One may have more than one node in the output layer, depending on the particular classification task at hand.

NEURAL NETWORKS- Number of Nodes

How many nodes should one have in the hidden layer?

Since more nodes in the hidden layer increases the power and flexibility of the network for identifying complex patterns, one might be tempted to have a large number of nodes in the hidden layer. On the other hand, an overly large hidden layer leads to over-fitting, memorizing the training set at the expense of generalizability to the validation set. If over-fitting is occurring, one may consider reducing the number of nodes in the hidden layer; conversely, if the training accuracy is unacceptably low, one may consider increasing the number of nodes in the hidden layer.

NEURAL NETWORKS- Number of Nodes

How many nodes should one have in the input layer?

The input layer accepts inputs from the data set, such as attribute values, and simply passes these values along to the hidden layer without further processing. Thus, the nodes in the input layer do not share the detailed node structure that the hidden layer nodes and the output layer nodes share.

NEURAL NETWORKS- Example

We will investigate the structure of hidden layer nodes and output layer nodes using the sample data provided in Figure 4.

$x_0 = 1.0$	$W_{0A} = 0.5$	$W_{0B} = 0.7$	$W_{0Z} = 0.5$
$x_1 = 0.4$	$W_{1A} = 0.6$	$W_{1B} = 0.9$	$W_{AZ} = 0.9$
$x_2 = 0.2$	$W_{2A} = 0.8$	$W_{2B} = 0.8$	$W_{BZ} = 0.9$
$x_3 = 0.7$	$W_{3A} = 0.6$	$W_{3B} = 0.4$	

Figure 4: Data Inputs and Initial Values for Neural Network Weights

NEURAL NETWORKS- Example

First, a combination function (usually summation, $\sum$) produces a linear combination of the node inputs and the connection weights into a single scalar value, which we will term *net*. Thus, for a given node j,

$$\text{net}_j = \sum_i W_{ij}x_{ij} = W_{0j}x_{0j} + W_{1j}x_{1j} + \cdots + W_{Ij}x_{Ij}$$

where x_{ij} represents the i^{th} input to node j, W_{ij} represents the weight associated with the i^{th} input to node j, and there are $I+1$ inputs to node j. Note that $x_1, x_2, \ldots, x_I$ represent inputs from upstream nodes, while x_0 represents a constant input, analogous to the constant factor in regression models, which by convention uniquely takes the value $x_{0j} = 1$. Thus, each hidden layer or output layer node j contains an "extra" input equal to a particular weight $W_{0j}x_{0j} = W_{0j}$, such as W_{0B} for node B.

NEURAL NETWORKS- Example

For example, for node A in the hidden layer, we have

$$\begin{aligned}\text{net}_A &= \sum_i W_{iA} x_{iA} = W_{0A}(1) + W_{1A}x_{1A} + W_{2A}x_{2A} + W_{3A}x_{3A} \\ &= 0.5 + 0.6(0.4) + 0.8(0.2) + 0.6(0.7) = 1.32\end{aligned}$$

Within node A, this combination function $\text{net}_A = 1.32$ is then used as an input to an activation function.

In biological neurons, signals are sent between neurons when the combination of inputs to a particular neuron cross a certain threshold, and the neuron “fires.”

This is nonlinear behavior, since the firing response is not necessarily linearly related to the increment in input stimulation. Artificial neural networks model this behavior through a **nonlinear activation function**.

NEURAL NETWORKS- Example

The most common activation function is the sigmoid function:

$$y = \frac{1}{1+e^{-x}}$$

where e is base of natural logarithms, equal to about 2.718281828. Thus, within node A, the activation would take $\text{net}_A = 1.32$ as input to the sigmoid activation function, and produce an output value of $y = \frac{1}{(1+e^{-1.32})} = 0.7892$.

Node A's work is done (for the moment), and this output value would then be passed along the connection to the output node Z, where it would form (via another linear combination) a component of net_Z.

NEURAL NETWORKS- Example

But before we can compute net_Z, we need to find the contribution of node B. From the values in Figure 4, we have,

$$\mathrm{net}_B = \sum_i W_{iB}x_{iB} = W_{0B}(1) + W_{1B}x_{1B} + W_{2B}x_{2B} + W_{3B}x_{3B}$$
$$= 0.7 + 0.9(0.4) + 0.8(0.2) + 0.4(0.7) = 1.5$$

Then,

$$f(\mathrm{net}_B) = \frac{1}{1+e^{-1.5}} = 0.8176$$

Node Z then combines these outputs from nodes A and B, through net_Z, a weighted sum, using the weights associated with the connections between these nodes.

NEURAL NETWORKS- Example

Note that the inputs x_i to node Z are not data attribute values but the outputs from the sigmoid functions from upstream nodes:

$$\begin{aligned}\text{net}_Z &= \sum_i W_{iZ}x_{iZ} = W_{0Z}(1) + W_{AZ}x_{AZ} + W_{BZ}x_{BZ} \\ &= 0.5 + 0.9(0.7892) + 0.9(0.8176) = 1.9461\end{aligned}$$

Finally, net_Z is input into the sigmoid activation function in node Z, resulting in

$$f(\text{net}_Z) = \frac{1}{1+e^{-1.9461}} = 0.8750$$

This value of 0.8750 is the output from the neural network for this first pass through the network, and represents the value predicted for the target variable for the first observation.

Sigmoid Activation Function

Why use the sigmoid function?

Because it combines nearly linear behavior, curvilinear behavior, and nearly constant behavior, depending on the value of the input.

Figure 5 shows the graph of the sigmoid function $y = f(x) = \frac{1}{(1+e^{-x})}$, for $-5 < x < 5$ [although $f(x)$ may theoretically take any real-valued input].

Through much of the center of the domain of the input x (e.g., $-1 < x < 1$), the behavior of $f(x)$ is nearly linear. As the input moves away from the center, $f(x)$ becomes curvilinear. By the time the input reaches extreme values, $f(x)$ becomes nearly constant.

Sigmoid Activation Function

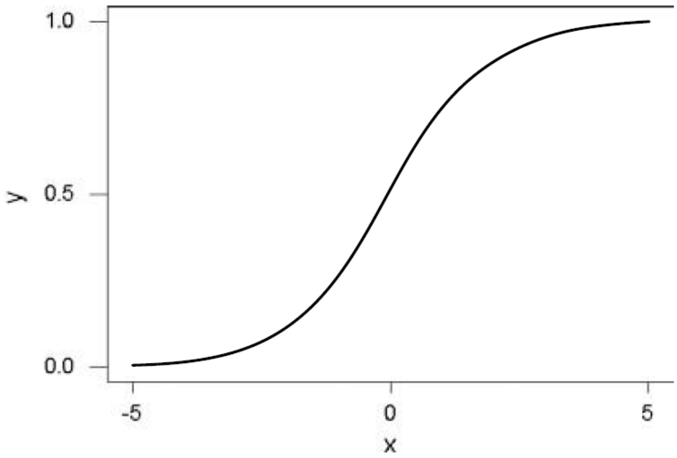


Figure 5: Graph of the sigmoid function $y = f(x) = \frac{1}{(1+e^{-x})}$.

BACK-PROPAGATION

How does the neural network learn?

Neural networks represent a supervised learning method, requiring a large training set of complete records, including the target variable. As each observation from the training set is processed through the network, an output value is produced from the output node (assuming that we have only one output node, as in Figure 3).

This output value is then compared to the actual value of the target variable for this training set observation, and the error (actual - output) is calculated. This prediction error is analogous to the residuals in regression models.

BACK-PROPAGATION- Squared Errors

To measure how well the output predictions fit the actual target values, most neural network models use the sum of squared errors:

$$\text{SSE} = \sum_{\text{records}} \sum_{\text{output nodes}} (\text{actual - output})^2$$

where the squared prediction errors are summed over all the output nodes and over all the records in the training set.

The problem is therefore to construct a set of model weights that will minimize the SSE. In this way, the weights are analogous to the parameters of a regression model. The “true” values for the weights that will minimize SSE are unknown, and our task is to estimate them, given the data. However, due to the nonlinear nature of the sigmoid functions permeating the network, there exists no closed-form solution for minimizing SSE as exists for least-squares regression.

GRADIENT DESCENT METHOD

We must therefore turn to optimization methods, specifically gradient-descent methods, to help us find the set of weights that will minimize SSE.

Suppose that we have a set (vector) of m weights $w = w_0, w_1, w_2, \ldots, w_m$ in our neural network model and we wish to find the values for each of these weights that, together, minimize SSE. We can use the gradient descent method, which gives us the direction that we should adjust the weights in order to decrease SSE. The gradient of SSE with respect to the vector of weights w is the vector derivative:

$$\nabla \text{SSE}(\text{W}) = \left[\frac{\partial \text{SSE}}{\partial w_0}, \frac{\partial \text{SSE}}{\partial w_1}, \ldots, \frac{\partial \text{SSE}}{\partial w_m} \right]$$

that is, the vector of partial derivatives of SSE with respect to each of the weights.

GRADIENT DESCENT METHOD- Working Method

To illustrate how gradient descent works, let us consider the case where there is only a single weight w_1. Consider Figure 6, which plots the error SSE against the range of values for w_1. We would prefer values of w_1 that would minimize the SSE.

The optimal value for the weight w_1 is indicated as w_1^*. We would like to develop a rule that would help us move our current value of w_1 closer to the optimal value w_1^* as follows:

$$w_{\text{new}} = w_{\text{current}} + \Delta w_{\text{current}}$$

where $\Delta w_{\text{current}}$ is the "change in the current location of w."

GRADIENT DESCENT METHOD- Working Method

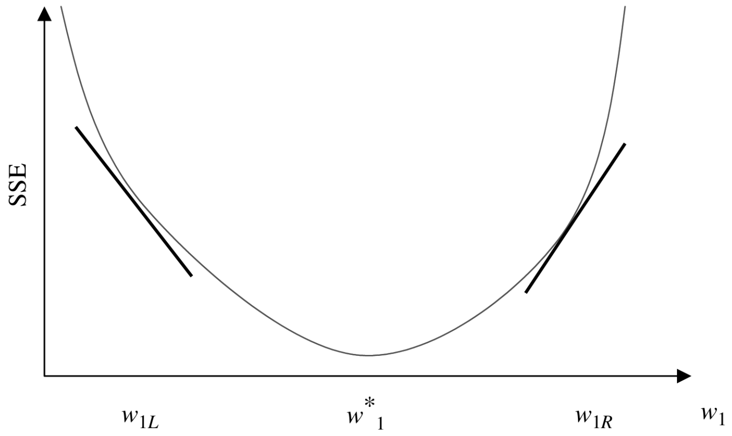


Figure 6: Using the slope of SSE with respect to w1 to find weight adjustment direction.

GRADIENT DESCENT METHOD- Working Method

Now, suppose that our current weight value $w_{current}$ is near w_{1L}. Then we would like to increase our current weight value to bring it closer to the optimal value w_1^* .

On the other hand, if our current weight value $w_{current}$ were near w_{1R} , we would instead prefer to decrease its value, to bring it closer to the optimal value w_1^*. Now the derivative $\partial \mathrm{SSE}/\partial w_1$ is simply the slope of the SSE curve at w_1.

For values of w_1 close to w_{1L} , this slope is negative, and for values of w_1 close to w_{1R} , this slope is positive. Hence, the direction for adjusting $w_{current}$ is the negative of the sign of the derivative of SSE at $w_{current}$, that is, -sign($\partial \mathrm{SSE}/\partial w_{current}$).

GRADIENT DESCENT METHOD- Working Method

Now, how far should $w_{current}$ be adjusted in the direction of -sign($\partial \mathrm{SSE}/\partial w_{current}$)?

Suppose that we use the magnitude of the derivative of SSE at $w_{current}$. When the curve is steep, the adjustment will be large, since the slope is greater in magnitude at those points. When the curve nearly flat, the adjustment will be smaller, due to less slope.

Finally, the derivative is multiplied by a positive constant η(Greek lowercase eta), called the *learning rate*, with values ranging between zero and 1.

The resulting form of $\Delta w_{current}$ is as follows: $\Delta w_{current} = -\eta(\partial \mathrm{SSE}/\partial w_{current})$, meaning that the change in the current weight value equals negative a small constant times the slope of the error function at $w_{current}$.

BACK-PROPAGATION RULES

The back-propagation algorithm takes the prediction error (actual - output) for a particular record and percolates the error back through the network, assigning partitioned responsibility for the error to the various connections. The weights on these connections are then adjusted to decrease the error, using gradient descent.

Using the sigmoid activation function and gradient descent, Mitchell [1] derives the back-propagation rules as follows:

$$w_{ij,new} = w_{ij,current} + \Delta w_{ij} \quad \text{where,} \quad \Delta w_{ij} = \eta \delta_j x_{ij}$$

BACK-PROPAGATION RULES

Now we know that η represents the learning rate and x_{ij} signifies the i^{th} input to node j, but what does δ_j represent? The component δ_j represents the responsibility for a particular error belonging to node j. The error responsibility is computed using the partial derivative of the sigmoid function with respect to net_j and takes the following forms, depending on whether the node in question lies in the output layer or the hidden layer:

$$\delta_j = \begin{cases} \text{output}_j(1-\text{output}_j)(\text{actual}_j-\text{output}_j) & \text{for output layer nodes} \\ \text{output}_j(1-\text{output}_j)\sum\limits_{\text{downstream}} W_{jk}\delta_j & \text{for hidden layer nodes} \end{cases}$$

where $\sum_{\text{downstream}} W_{jk}\delta_j$ refers to the weighted sum of the error responsibilities for the nodes downstream from the particular hidden layer node. (For the full derivation, see Mitchell [1].)

BACK-PROPAGATION RULES

Also, note that the back-propagation rules illustrate why the attribute values need to be normalized to between zero and 1.

For example, if income data, with values ranging into six figures, were not normalized, the weight adjustment $\Delta w_{ij} = \eta \delta_j x_{ij}$ would be dominated by the data value x_{ij}. Hence the error propagation (in the form of δ_j) through the network would be overwhelmed, and learning (weight adjustment) would be stifled.

EXAMPLE OF BACK-PROPAGATION

Recall from our introductory example that the output from the first pass through the network was *output* = 0.8750. Assume that the actual value of the target attribute is *actual* = 0.8 and that we will use a *learning rate* of $\eta = 0.01$. Then the *prediction error* equals $0.8 - 0.8750 = -0.075$, and we may apply the foregoing rules to illustrate how the back-propagation algorithm works to adjust the weights by portioning out responsibility for this error to the various nodes.

Although it is possible to update the weights only after all records have been read, neural networks use stochastic (or online) back-propagation, which updates the weights after each record.

EXAMPLE OF BACK-PROPAGATION

First, the error responsibility δ_Z for node Z is found. Since node Z is an output node, we have

$$\begin{aligned}\delta_Z &= \text{output}_Z(1-\text{output}_Z)(\text{actual}_Z-\text{output}_Z)\\ &= 0.875(1-0.875)(0.8-0.875) = -0.0082\end{aligned}$$

We may now adjust the **"constant"** weight W_{0Z}(which transmits an "input" of 1) using the back-propagation rules as follows:

$$\Delta W_{0Z} = \eta\,\delta_Z(1) = 0.1(-0.0082)(1) = -0.00082$$
$$w_{0Z,new} = w_{0Z,current} + \Delta w_{0Z} = 0.5 - 0.00082 = 0.49918$$

EXAMPLE OF BACK-PROPAGATION

Next, we move upstream to node A. Since node A is a hidden layer node, its error responsibility is

$$\delta_A = \text{output}_A(1 - \text{output}_A) \sum_{\text{downstream}} W_{jk}\delta_A$$

The only node downstream from node A is node Z . The weight associated with this connection is $W_{AZ} = 0.9$, and the error responsibility at node Z is -0.0082, so that $\delta_A = 0.7892(1 - 0.7892)(0.9)(-0.0082) = -0.00123$.

We may now update weight W_{AZ} using the back-propagation rules as follows:

$$\Delta W_{AZ} = \eta\,\delta_Z.\text{output}_A = 0.1(-0.0082)(0.7892) = -0.000647$$
$$w_{AZ,new} = w_{AZ,current} + \Delta w_{AZ} = 0.9 - 0.000647 = 0.899353$$

The weight for the connection between hidden layer node A and output layer node Z has been adjusted from its initial value of 0.9 to its new value of 0.899353.

EXAMPLE OF BACK-PROPAGATION

Next, we turn to node B, a hidden layer node, with error responsibility

$$\delta_B = \text{output}_B(1 - \text{output}_B) \sum_{\text{downstream}} W_{jk}\delta_A$$

Again, the only node downstream from node B is node Z,giving us $\delta_B = 0.8176(1 - 0.8176)(0.9)(-0.0082) = -0.0011$.

Weight W_{BZ} may then be adjusted using the back-propagation rules as follows:

$$\Delta W_{BZ} = \eta\,\delta_Z.\text{output}_B = 0.1(-0.0082)(0.8176) = -0.00067$$
$$w_{BZ,new} = w_{BZ,current} + \Delta w_{BZ} = 0.9 - 0.00067 = 0.89933$$

We move upstream to the connections being used as inputs to node A.

EXAMPLE OF BACK-PROPAGATION

For weight W_{1A} we have

$$\Delta W_{1A} = \eta \delta_A x_1 = 0.1(-0.00123)(0.4) = -0.0000492$$
$$w_{1A,new} = w_{1A,current} + \Delta w_{1A} = 0.6 - 0.0000492 = 0.5999508.$$

For weight W_{2A} we have

$$\Delta W_{2A} = \eta \delta_A x_2 = 0.1(-0.00123)(0.2) = -0.0000246$$
$$w_{2A,new} = w_{2A,current} + \Delta w_{2A} = 0.8 - 0.0000246 = 0.7999754.$$

For weight W_{3A} we have

$$\Delta W_{3A} = \eta \delta_A x_3 = 0.1(-0.00123)(0.7) = -0.0000861$$
$$w_{3A,new} = w_{3A,current} + \Delta w_{3A} = 0.6 - 0.0000861 = 0.5999139.$$

Finally, for weight W_{0A} we have

$$\Delta W_{0A} = \eta \delta_A (1) = 0.1(-0.00123) = -0.000123$$
$$w_{0A,new} = w_{0A,current} + \Delta w_{0A} = 0.5 - 0.000123 = 0.499877.$$

EXAMPLE OF BACK-PROPAGATION

Adjusting weights W_{0B}, W_{1B}, W_{2B}, and W_{3B} is left as an exercise.

Note that the weight adjustments have been made based on only a single perusal of a single record. The network calculated a predicted value for the target variable, compared this output value to the actual target value, and then percolated the error in prediction throughout the network, adjusting the weights to provide a smaller prediction error.

Showing that the adjusted weights result in a smaller prediction error is left as an exercise.

TERMINATION CRITERIA

The neural network algorithm would then proceed to work through the training data set, record by record, adjusting the weights constantly to reduce the prediction error.

It may take many passes through the data set before the algorithm's termination criterion is met. What, then, serves as the termination criterion, or stopping criterion? If training time is an issue, one may simply set the number of passes through the data, or the amount of real-time the algorithm may consume, as termination criteria.

However, what one gains in short training time is probably bought with degradation in model efficacy.

TERMINATION CRITERIA

Alternatively, one may be tempted to use a termination criterion that assesses when the SSE on the training data has been reduced to some low threshold level.

Unfortunately, because of their flexibility, neural networks are prone to over-fitting, memorizing the idiosyncratic patterns in the training set instead of re-taining generalizability to unseen data.

TERMINATION CRITERIA

Therefore, most neural network implementations adopt the following cross-validation termination procedure:

1. Retain part of the original data set as a holdout validation set.
2. Proceed to train the neural network as above on the remaining training data.
3. Apply the weights learned from the training data on the validation data.
4. Monitor two sets of weights, one “current” set of weights produced by the training data, and one “best” set of weights, as measured by the lowest SSE so far on the validation data.
5. When the current set of weights has significantly greater SSE than the best set of weights, then terminate the algorithm.

TERMINATION CRITERIA

Regardless of the stopping criterion used, the neural network is not guaranteed to arrive at the optimal solution, known as the *global minimum* for the SSE. Rather, the algorithm may become stuck in a local minimum, which represents a good, if not optimal solution. In practice, this has not presented an insuperable problem.

- For example, multiple networks may be trained using different initialized weights, with the best-performing model being chosen as the “final” model.
- Second, the *online* or *stochastic back-propagation* method itself acts as a guard against getting stuck in a local minimum, since it introduces a random element to the gradient descent (see Reed and Marks [2]).
- Alternatively, a *momentum* term may be added to the back-propagation algorithm, with effects discussed below.

Learning Rate

Recall that the learning rate η, $0 < \eta < 1$, is a constant chosen to help us move the network weights toward a global minimum for SSE.

However, what value should η take? How large should the weight adjustments be?

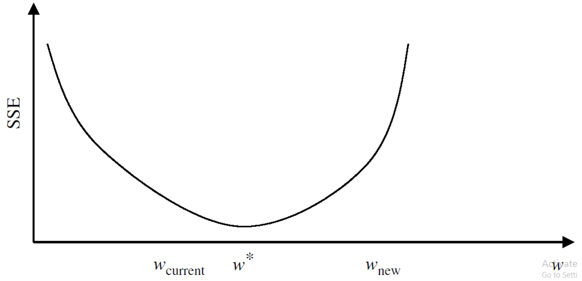


Figure 7: Large η may cause algorithm to overshoot global minimum.

Learning Rate

When the learning rate is very small, the weight adjustments tend to be very small. Thus, if η is small when the algorithm is initialized, the network will probably take an unacceptably long time to converge. Is the solution therefore to use large values for η? Not necessarily. Suppose that the algorithm is close to the optimal solution and we have a large value for η. This large η will tend to make the algorithm overshoot the optimal solution.

Consider Figure 7, where W^* is the optimum value for weight W, which has current value $W_{current}$. According to the gradient descent rule, $\Delta w_{current} = -\eta(\partial SSE/\partial w_{c}$ $W_{current}$ will be adjusted in the direction of W^*. But if the learning rate η, which acts as a multiplier in the formula for $\Delta w_{current}$, is too large, the new weight value W_{new} will jump right past the optimal value W^*, and may in fact end up farther away from W^* than $W_{current}$.

Learning Rate

In fact, since the new weight value will then be on the opposite side of W^*, the next adjustment will again overshoot W^*, leading to an unfortunate oscillation between the two “slopes” of the valley and never settling down in the ravine (the minimum).

One solution is to allow the learning rate η to change values as the training moves forward.

At the start of training, η should be initialized to a relatively large value to allow the network to quickly approach the general neighborhood of the optimal solution. Then, when the network is beginning to approach convergence, the learning rate should gradually be reduced, thereby avoiding overshooting the minimum.

MOMENTUM

The back-propagation algorithm is made more powerful through the addition of a *momentum term*, α, as follows:

$$\Delta w_{current} = -\eta \frac{\partial \mathrm{SSE}}{\partial w_{current}} + \alpha \Delta w_{previous}$$

where $\Delta w_{previous}$ represents the previous weight adjustment, and $0 \leq \alpha < 1$. Thus, the new component $\alpha \Delta w_{previous}$ represents a fraction of the previous weight adjustment for a given weight.

MOMENTUM

Essentially, the momentum term represents inertia. Large values of α will influence the adjustment in the current weight, $\Delta w_{current}$, to move in the same direction as previous adjustments. It has been shown (e.g., Reed and Marks [2]) that including momentum in the back-propagation algorithm results in the adjustment becoming an exponential average of all previous adjustments:

$$\Delta w_{current} = -\eta \sum_{k=0}^{\text{inf}} \alpha^k \frac{\partial \text{SSE}}{\partial w_{current-k}}$$

The α^k term indicates that the more recent adjustments exert a larger influence. Large values of α allow the algorithm to "remember" more terms in the adjustment history. Small values of α reduce the inertial effects as well as the influence of previous adjustments, until, with $\alpha = 0$, the component disappears entirely.

MOMENTUM

Clearly, a momentum component will help to dampen the oscillations around optimality mentioned earlier, by encouraging the adjustments to stay in the same direction. But momentum also helps the algorithm in the early stages of the algorithm, by increasing the rate at which the weights approach the neighborhood of optimality. This is because these early adjustments will probably all be in the same direction, so that the exponential average of the adjustments will also be in that direction. Momentum is also helpful when the gradient of SSE with respect to w is flat.

If the momentum term α is too large, however, the weight adjustments may again overshoot the minimum, due to the cumulative influences of many previous adjustments.

MOMENTUM

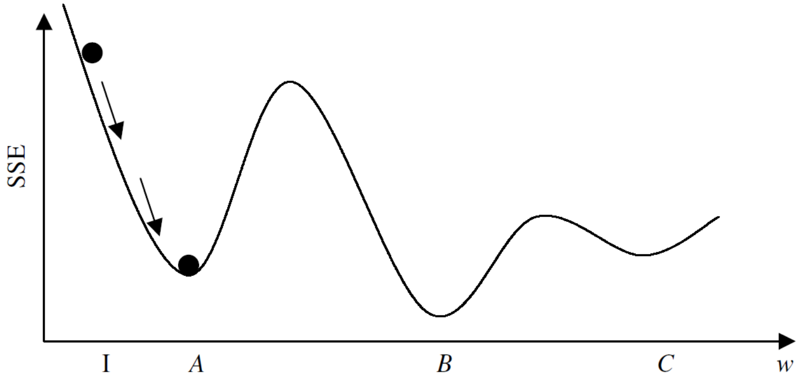


Figure 8: Small momentum α may cause algorithm to undershoot global minimum.

MOMENTUM

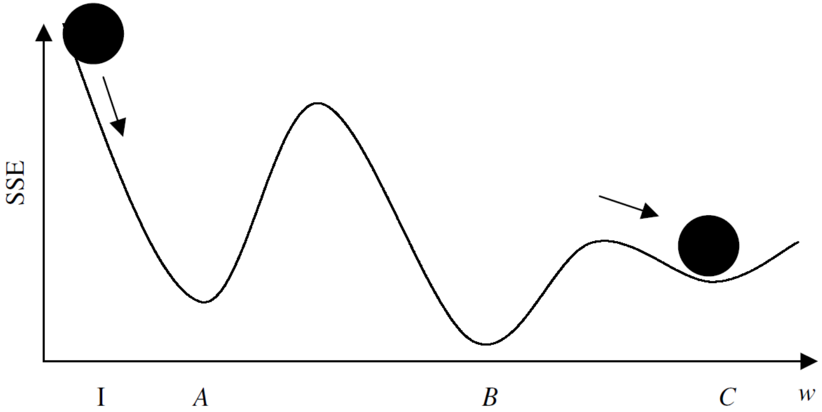


Figure 9: Large momentum α may cause algorithm to overshoot global minimum.

MOMENTUM

For an informal appreciation of momentum, consider Figures 8 and 9. In both figures, the weight is initialized at location I, local minima exist at locations A and C, with the optimal global minimum at B.

In Figure 8, suppose that we have a small value for the momentum term α, symbolized by the small mass of the “ball” on the curve. If we roll this small ball down the curve, it may never make it over the first hill, and remain stuck in the first valley. That is, the small value for α enables the algorithm to easily find the first trough at location A, representing a local minimum, but does not allow it to find the global minimum at B.

MOMENTUM

Next, in Figure 9, suppose that we have a large value for the momentum term 9, symbolized by the large mass of the “ball” on the curve. If we roll this large ball down the curve, it may well make it over the first hill but may then have so much momentum that it overshoots the global minimum at location B and settles for the local minimum at location C.

Thus, one needs to consider carefully what values to set for both the learning rate η and the momentum term α. Experimentation with various values of η and α may be necessary before the best results are obtained.

SENSITIVITY ANALYSIS

One of the drawbacks of neural networks is their opacity. The same wonderful flexibility that allows neural networks to model a wide range of nonlinear behavior also limits our ability to interpret the results using easily formulated rules. Unlike decision trees, no straightforward procedure exists for translating the weights of a neural network into a compact set of decision rules.

However, a procedure is available, called *sensitivity analysis*, which does allow us to measure the relative influence each attribute has on the output result. Using the test data set mentioned above, the sensitivity analysis proceeds as follows:

1. Generate a new observation x_{mean}, with each attribute value in x_{mean} equal to the mean of the various attribute values for all records in the test set.
2. Find the network output for input x_{mean}. Call it $\text{output}_{\text{mean}}$.
3. Attribute by attribute, vary x_{mean} to reflect the attribute minimum and maximum. Find the network output for each variation and compare it to $\text{output}_{\text{mean}}$.

SENSITIVITY ANALYSIS

The sensitivity analysis will find that varying certain attributes from their minimum to their maximum will have a greater effect on the resulting network output than it has for other attributes. For example, suppose that we are interested in predicting stock price based on *price–earnings ratio*, *dividend yield*, and other attributes. Also, suppose that varying *price–earnings* ratio from its minimum to its maximum results in an increase of 0.20 in the network output, while varying *dividend yield* from its minimum to its maximum results in an increase of 0.30 in the network output when the other attributes are held constant at their mean value.

We conclude that the network is more *sensitive* to variations in dividend yield and that therefore dividend yield is a more important factor for predicting stock prices than is price–earnings ratio.

APPLICATION OF NEURAL NETWORK MODELING

Next, we apply a neural network model using Insightful Miner on the same adult data set [3] from the UCal Irvine Machine Learning Repository that we analyzed in Lecture 6. The Insightful Miner neural network software was applied to a training set of 24,986 cases, using a single hidden layer with eight hidden nodes. The algorithm iterated 47 epochs (runs through the data set) before termination. The resulting neural network is shown in Figure 10.

The squares on the left represent the input nodes. For the categorical variables, there is one input node per class. The eight dark circles represent the hidden layer. The light gray circles represent the constant inputs. There is only a single output node, indicating whether or not the record is classified as having income less than $50,000.

APPLICATION OF NEURAL NETWORK MODELING

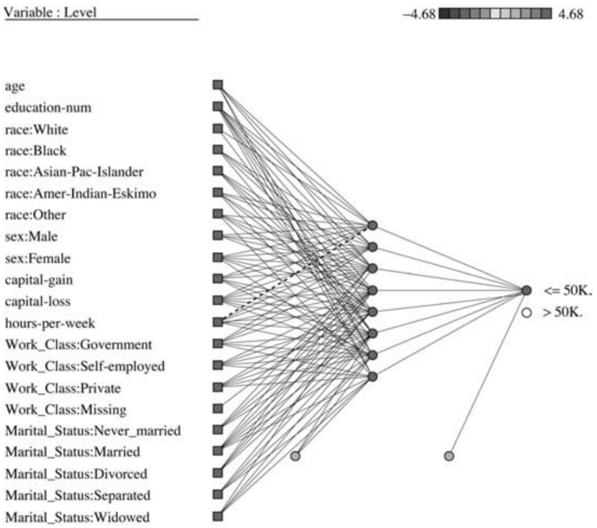
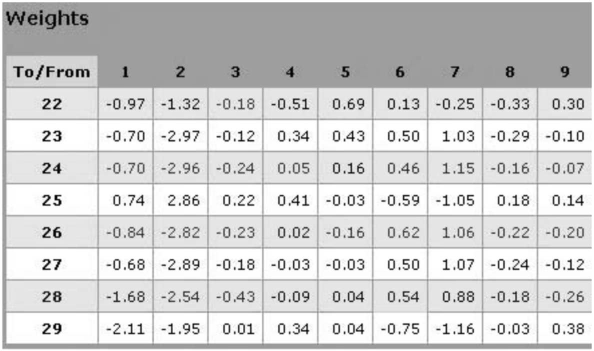


Figure 10: Neural network for the adult data set generated by Insightful Miner.

APPLICATION OF NEURAL NETWORK MODELING

In this algorithm, the weights are centered at zero. An excerpt of the computer output showing the weight values is provided in Figure 11. The columns in the first table represent the input nodes: 1 = age, 2 = education-num, and so on, while the rows represent the hidden layer nodes: 22 = first (top) hidden node, 23 = second hidden node, and so on. For example, the weight on the connection from age to the topmost hidden node is -0.97, while the weight on the connection from Race: American Indian/Eskimo (the sixth input node) to the last (bottom) hidden node is -0.75. The Figure 12 displays the weights from the hidden nodes to the output node.

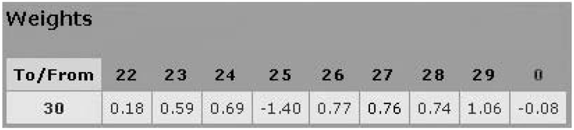
APPLICATION OF NEURAL NETWORK MODELING



Weights

To/From	1	2	3	4	5	6	7	8	9
22	-0.97	-1.32	-0.18	-0.51	0.69	0.13	-0.25	-0.33	0.30
23	-0.70	-2.97	-0.12	0.34	0.43	0.50	1.03	-0.29	-0.10
24	-0.70	-2.96	-0.24	0.05	0.16	0.46	1.15	-0.16	-0.07
25	0.74	2.86	0.22	0.41	-0.03	-0.59	-1.05	0.18	0.14
26	-0.84	-2.82	-0.23	0.02	-0.16	0.62	1.06	-0.22	-0.20
27	-0.68	-2.89	-0.18	-0.03	-0.03	0.50	1.07	-0.24	-0.12
28	-1.68	-2.54	-0.43	-0.09	0.04	0.54	0.88	-0.18	-0.26
29	-2.11	-1.95	0.01	0.34	0.04	-0.75	-1.16	-0.03	0.38

Figure 11: Some of the neural network weights for the income example.



Weights

To/From	22	23	24	25	26	27	28	29	0
30	0.18	0.59	0.69	-1.40	0.77	0.76	0.74	1.06	-0.08

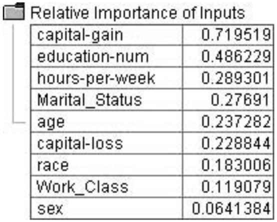
Figure 12: Some of the neural network weights for the income example.

APPLICATION OF NEURAL NETWORK MODELING

The estimated prediction accuracy using this very basic model is 82%, which is in the ballpark of the accuracies reported by Kohavi [4]. Since over 75subjects have incomes below \$50,000, simply predicted “less than \$50,000” for every person would provide a baseline accuracy of about 75%.

However, we would like to know which variables are most important for predicting (classifying) income. We therefore perform a sensitivity analysis using Clementine, with results shown in Figure 13. Clearly, the amount of capital gains is the best predictor of whether a person has income less than \$50,000, followed by the number of years of education. Other important variables include the number of hours worked per week and marital status. A person’s gender does not seem to be highly predictive of income.

APPLICATION OF NEURAL NETWORK MODELING


Relative Importance of Inputs

capital-gain	0.719519
education-num	0.486229
hours-per-week	0.289301
Marital_Status	0.27691
age	0.237282
capital-loss	0.228844
race	0.183006
Work_Class	0.119079
sex	0.0641384

Figure 13: Most important variables: results from sensitivity analysis.

Of course, there is much more involved with developing a neural network classification model. For example, further data preprocessing may be called for; the model would need to be validated using a holdout validation data set, and so on. For a start-to-finish application of neural networks to a real-world data set, from data preparation through model building and sensitivity analysis, see Reference [5].

Further Reading

- Chapter 4.6 and 6.3 of Data Mining - Practical Machine Learning Tools and Techniques, Second Edition - Ian H. Witten, Eibe Frank
- Chapter 7 of DISCOVERING KNOWLEDGE IN DATA - An Introduction to Data Mining - DANIEL T. LAROSE
- Chapter 5.4 of Introduction to Data Mining (Second Edition) - Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, Vipin Kumar

References

[1] T. Mitchell, *Machine Learning*, ser. McGraw-Hill series in computer science. McGraw-Hill, 1997. [Online]. Available: https://books.google.com.bd/books?id=B9gZygEACAAJ

[2] R. D. Reed and R. J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998.

[3] C. B. D. Newman and C. Merz, "UCI repository of machine learning databases," 1998. [Online]. Available: http://www.ics.uci.edu/$\sim$mlearn/MLRepository.html

[4] R. Kohavi, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid," 12 1996. [Online]. Available: https://www.osti.gov/biblio/421279

[5] D. T. Larose, "Data mining methods and models," p. 1–32, 2006.

Thank you.
Any Questions?