

Artificial Neural Network for Spam Email Classification

Ian R. Stewart

November 26th, 2018

Fall 2018 – COSC 528

Project 4

Abstract

An artificial neural network was created to predict the classification, either spam or not spam, of an email containing specific features. A neural network class, called *ANN()*, was created in Python and allows the user to create a unique network architecture through input eight parameters: (1) number of inputs, (2) number of hidden layers, (3) number of neurons per hidden layer, (4) number of inputs, (5) activation function, (6) learning rate, (7) number of epochs, and (8) training-test split percentage. The *ANN()* begins by randomly initializing the weights between the network layers and employs forward propagation to calculate the predicted output classification. The weights are updated online using the back-propagation error method in reverse order, from the output to input layer. This process is completed until the number of epochs are completed or a specified error convergence rate is met. The neural networks using sigmoid and linear activation functions were assessed via altering two hyperparameters: number of the hidden layers and the number of neurons per hidden layer. Using all 57 data features for the input layer (i.e. 57 nodes on the input layer) and two neurons for the output, the networks were evaluated by calculating the accuracy and F1 score of the network as the two hyperparameters are iteratively increased. The number of hidden layers chosen was three as the error beyond three layers contained variations (increased and decreased errors), which is expected as model complexity increases resulting in overfitting of the data. The number of hidden neurons to used were three and six, respectively, for the sigmoid and linear activation functions. The data was reduced using PCA to 41 features that comprised 95% of the total variance explained. The *ANN()* for the new, lower dimensional dataset resulted in comparatively similar accuracies and F1 scores for both activation functions with only slight decreases.

Table of Contents

1 Introduction.....	4
2 Data Exploration	6
3 Data Analysis	9
3.1 Results.....	12
3.2 Dimensionality Reduction Analysis	19
References.....	22

List of Tables

Table 1: Email dataset feature labels and data types.	6
Table 2: Data exploration for amount of zero-value entries in data features.....	7
Table 3: Performance metrics for ANN with multiple hidden layers using sigmoid function.	15
Table 4: Performance metrics for ANN with multiple hidden layers using linear function.....	16
Table 5: Performance metrics for ANN changing number of neuron with sigmoid function	17
Table 6: Performance metrics for ANN changing number of neuron with linear function.....	18

List of Figures

Figure 1: Example three-layer neural network.	4
Figure 2: Order of dataset with respect to classification.....	8
Figure 3: Logistic sigmoid activation function.	10
Figure 4: Linear activation function.	10
Figure 5: Softmax activation function.	10
Figure 6: Forward Propagation output for example network with a sigmoid activation function.....	11
Figure 7: General formula for calculating error in ANN.....	11
Figure 8: Pseudocode for back propagation.....	12
Figure 9: Formula to update network weights based on error and learning rate.	12
Figure 10: Screenshot of iPython Notebook for created <code>ANN()</code> class using sigmoid function....	13
Figure 11: Training error for initial <code>ANN()</code> class run with sigmoid function.....	13
Figure 12: Screenshot of iPython Notebook for created <code>ANN()</code> class using linear function.	14
Figure 13: Training error for initial <code>ANN()</code> class run with linear function.	15
Figure 14: ANN accuracy with sigmoid function for changing number of hidden layers.	16
Figure 15: ANN accuracy with linear function for changing number of hidden layers.	16
Figure 16: ANN accuracy using sigmoid function with changing number of neurons on three hidden layers.	18
Figure 17: ANN accuracy using linear function with changing number of neurons on three hidden layers.	19
Figure 18: Variance explained for each singular value using PCA.	20
Figure 19: Screenshot of PCA data for ANN with sigmoid function.	21
Figure 20: Screenshot of PCA data for ANN with linear function.....	21

1 Introduction

In this project, an artificial neural network is created using a backpropagation algorithm to classify an email with specific features as either spam or non-spam (i.e. binary classification). Artificial neural networks are ideally suited for classification problems, where the analysis goal is to classify a data entry based on a particular set data attributes or features.¹ Neural networks are used to transform the data into a relatively complex set of nonlinear basis functions or decision boundaries. A network is organized into layers with three general categories: input layer, output layer, and hidden layer. A user can choose to employ multiple hidden layers, in which the user can choose an optimal number of layers by iteratively tuning the configuration while training the network by using ancillary algorithms. Multiple hidden layers allow for representation of nonlinear relationships between the input and output layers to be accounted for (i.e. accounts for nonlinearly separable data).

The backpropagation algorithm is categorized to as a feedforward neural network, which means the information flows in a single direction from the input layer to the output layer, absent of any network loops or cycles. This approach is a supervised learning technique that essentially predicts the classification by modifying the input weights from the previous layer if using multilayer or from the initial input layer to produce an expected classification. The weights are calculated using a training dataset using fully-connected layers. To add context to this approach, a single hidden layer network is shown in the following Figure 1, where one input layer with a length equal to the number of data features and an output layer with the length equal to the number of classes (i.e. two classes in the spam versus non-spam binary classification problem).

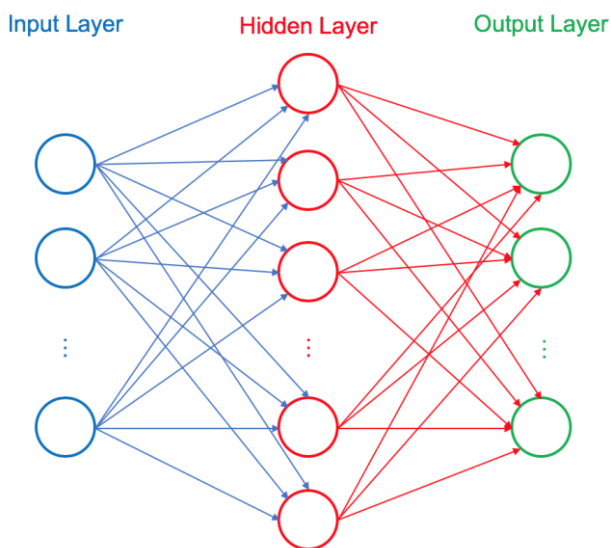


Figure 1: Example three-layer neural network.

¹ Another common problem set for neural networks to solve is regression, where a neuron activation is obtained by the weight sum of the inputs for each layers, similar to linear regression.

Using the two-class (binary) classification for the spam or non-spam problem, the outputs layer can be hot-coded by vectorizing the output into a single column vector where [0,1] or [1,0] can represent spam and non-spam classification, respectively.

The backpropagation algorithm can be organized into five steps:

- 1. Initialize neural network;**
 - *Description:* Create network architecture using predetermined number of hidden layers and number of neurons per layer and initialize weights.
- 2. Forward propagate weight;**
 - *Description:* Propagate input signal through the hidden layers through until the output layer is reached. From Figure 1, this step essentially propagates the input features through the connection from left (input) to right (output).
- 3. Back propagate error;**
 - *Description:* Determine difference, or error, between expected output and calculated forward propagate output. From Figure 1, this step propagates the error from right (output) to the left (input).
- 4. Train network on training data; and,**
 - *Description:* Iterate randomly through the dataset to train the network by updating the weights via the weights from each iteration.
- 5. Predict classification based on calculated layer weights.**
 - *Description:* Use the calculated weights for the neural network to predict the classification of the test data.

The development and implementation of this algorithm in Python follows these steps to predict the classification for a set of test emails. Discussion regarding the creation and implementation of the Python functions are provided in the latter portion of the report (see 3 Data Analysis section).

2 Data Exploration

The dataset utilized in this project contains 4,601 entries and 57 total columns or features with an additional feature containing the actual classification value. The data was found to be full with no null and absent of negative values in each feature, which is expected. Table 1 provides the data feature labels and the corresponding data types.

Table 1: Email dataset feature labels and data types.

Feature Label	Data Type		Feature Label	Data Type
word_freq_make	float64		word_freq_labs	float64
word_freq_address	float64		word_freq_telnet	float64
word_freq_all	float64		word_freq_857	float64
word_freq_3d	float64		word_freq_data	float64
word_freq_our	float64		word_freq_415	float64
word_freq_over	float64		word_freq_85	float64
word_freq_remove	float64		word_freq_technology	float64
word_freq_internet	float64		word_freq_1999	float64
word_freq_order	float64		word_freq_parts	float64
word_freq_mail	float64		word_freq_pm	float64
word_freq_receive	float64		word_freq_direct	float64
word_freq_will	float64		word_freq_cs	float64
word_freq_people	float64		word_freq_meeting	float64
word_freq_report	float64		word_freq_original	float64
word_freq_addresses	float64		word_freq_project	float64
word_freq_free	float64		word_freq_re	float64
word_freq_business	float64		word_freq_edu	float64
word_freq_email	float64		word_freq_table	float64
word_freq_you	float64		word_freq_conference	float64
word_freq_credit	float64		char_freq_;	float64
word_freq_your	float64		char_freq_(float64
word_freq_font	float64		char_freq_[float64
word_freq_000	float64		char_freq_!	float64
word_freq_money	float64		char_freq_\$	float64
word_freq_hp	float64		char_freq_#	float64
word_freq_hpl	float64		capital_run_length_average	float64
word_freq_george	float64		capital_run_length_longest	int64
word_freq_650	float64		capital_run_length_total	int64
word_freq_lab	float64		spam class	int64

The minimum value for all data is equal to 0.0, except for the three *capital_run_length** features, which have a minimum value of 1.0. The maximum values are unique for each feature, which is expected as certain features are traditionally more common than others, such as the features *word_freq_you* versus *word_freq_1999*. The amount of 0.0 or no-value could also be of interest to evaluate if certain features do not contain much data. This analysis is provided in Table 2.

Table 2: Data exploration for amount of zero-value entries in data features.

Feature	# no-values	Feature	# no-values
word_freq_make	3548	word_freq_labs	4132
word_freq_address	3703	word_freq_telnet	4308
word_freq_all	2713	word_freq_857	4396
word_freq_3d	4554	word_freq_data	4196
word_freq_our	2853	word_freq_415	4386
word_freq_over	3602	word_freq_85	4116
word_freq_remove	3794	word_freq_technology	4002
word_freq_internet	3777	word_freq_1999	3772
word_freq_order	3828	word_freq_parts	4518
word_freq_mail	3299	word_freq_pm	4217
word_freq_receive	3892	word_freq_direct	4148
word_freq_will	2276	word_freq_cs	4453
word_freq_people	3749	word_freq_meeting	4260
word_freq_report	4244	word_freq_original	4226
word_freq_addresses	4265	word_freq_project	4274
word_freq_free	3360	word_freq_re	3290
word_freq_business	3638	word_freq_edu	4084
word_freq_email	3563	word_freq_table	4538
word_freq_you	1374	word_freq_conference	4398
word_freq_credit	4177	char_freq_;	3811
word_freq_your	2178	char_freq_(1886
word_freq_font	4484	char_freq_[4072
word_freq_000	3922	char_freq_!	2343
word_freq_money	3866	char_freq_\$	3201
word_freq_hp	3511	char_freq_#	3851
word_freq_hpl	3790	capital_run_length_average	0
word_freq_george	3821	capital_run_length_longest	0
word_freq_650	4138	capital_run_length_total	0
word_freq_lab	4229		

Outside of the *capital_run_length** features, the maximum zero-valued entry *word_freq_3d* was 4554 (98.7% of total) and the minimum zero-valued entry *word_freq_you* was 1374 (29.8% of total), which is not an extremely surprising result as *3d* is not as common as term *you* in general conversations. Moreover, 37 of the 46 features are equal to or greater 80% zero-valued entries. A

value of zero (i.e. representing no term in email) is not necessarily a negative attribute, but a large amount of no values could provide strong correlations for classification or dimensionality reduction performed later. Moving forward, the data features are further delineated by the following unique attributes in order in the dataset.

- First 48 features are continuous real values ranging $[0,100]$ that represent the percentage of words in the email that match the specific word (i.e. `word_freq_WORD`).
- Next 6 features are continuous real values ranging $[0,100]$ that represent the percentage of characters in the email that match the specific character (i.e. `word_freq_CHARACTER`).
- The next feature, *capital_run_length_average*, is continuous real value ranging $[1, \dots]$ that represents the average length of uninterrupted sequences of capital letters.
- The next feature, *capital_run_length_longest*, is continuous real value ranging $[1, \dots]$ that represents the longest of uninterrupted sequence of capital letters in the email.
- The next feature, *capital_run_length_total*, is continuous real value ranging $[1, \dots]$ that represents the sum of length of uninterrupted sequences of capital letters in the email.
- The final feature, *spam class*, denotes whether the email was considered spam (1) or non-spam (0).

Another observation is that the classifications are actually grouped together, where the spam classifications are the first 1,813 entries and the latter portion of the data (i.e. 2,788 entries) are all non-spam classification. This is an interesting finding, as a small training-test split for a cross-validation study can potentially cause classification errors if a sufficient amount of each group is not represented in the training set. Being a two-class dataset, the first portion of the data (~40%) represent only spam email classifications and the latter portion (60%) of the data represents only non-spam email classifications, as illustrated in the following Figure 2.

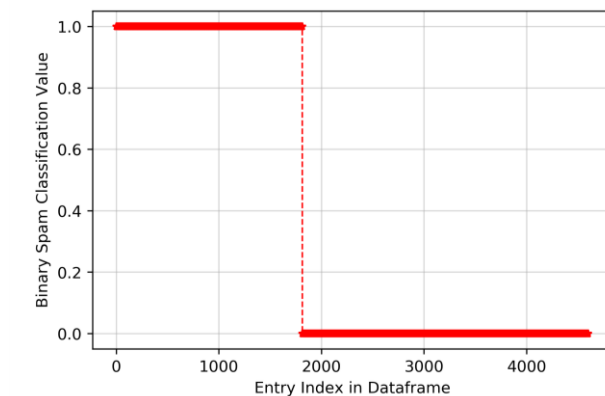


Figure 2: Order of dataset with respect to classification.

3 Data Analysis

When creating a neural network, several questions must be answered to building and optimizing the network, namely:

1. How many hidden layers to employ to obtain an optimal predictive power?
2. How many neurons per hidden layer is optimal?
3. How to obtain answers to the previous two question?

Only one input layer is used for a network with the number of inputs based on the length of the known data features or metrics. Similarly, in this case the output layer is also known beforehand and represents the number of classes (i.e. two). These questions will be addressed as the data analysis is performed in this section.

The network is first initialized by the created *initialize()* Python function within the *ANN* class, which accepts the number of inputs in the input layer, the number of neurons in the hidden layers, and the number of hidden layers. As each hidden layer has a set number of neurons, the weights must be maintained for the input and the bias. To store this information, the created *initialize()* Python function stores the data in a dictionary that holds the network weights for each layer in order (i.e. begins with the weights between the input layer and the first hidden layer and ends with the weights between the final hidden layers and the output layer). During the initialization step, the weights are randomly selected from a uniform distribution ranging from 0 to 1 using the Python library *random* via the function *random.uniform()* and the number of weight inputs for the hidden layers is one greater than the specific value of neurons to account for the bias at each layer, which is initially fixed to a value of 1.0. Before moving on the following steps described in the Introduction section, let's examine the output for an initialized network with two inputs in the input layer, two hidden layers with three neurons per layer, and two outputs in the output layer. The *initialize()* function provides the following output from an array of length equal to the number of weights between each step (i.e. three in this example).

- Weights between input and first hidden layer:

```
[
    {'weights': [0.7270700037509241, 0.7383627445080684, 0.18978193
074880423]},
    {'weights': [0.7869285898008361, 0.11191242288793657, 0.6323166
547815777]},
    {'weights': [0.5542929997136264, 0.6299108552149877, 0.03687222
4253109986]}
]
```

- Weights between hidden layers:

```
[
    {'weights': [0.9541070703271004, 0.9002708967220981, 0.55742110
04774138, 0.56239456051497]},
    {'weights': [0.5772579431172555, 0.5744375394342741, 0.27621222
9019259, 0.5615663370673547]},
    {'weights': [0.49917371980390934, 0.2289715704548373, 0.9165265
896818503, 0.4831653543316994]}
]
```

]

- Weights between last (second) hidden layer and output layer:

```
[
    {'weights': [0.7936276360327776, 0.9247539681130554, 0.0889947
9960010304, 0.6728626611317721]},
    {'weights': [0.19170994036771682, 0.8864472297353635, 0.120122
5021016884, 0.15257603530797048]}
]
```

From the created arrays, the hidden layers weights contain three neurons, indicated by the number of Python dictionaries in the array, and three weights for each neuron plus one weight the bias resulting in a total of four weights. The output layer array contains two Python dictionaries with a total of four weights, three of which correspond to the number of neurons and an additional weight for the bias. The length of the output array (i.e. two) is equal to the number of output neurons on the output layer.

Upon initialization of the weights, the forward-propagation step is performed. In this step, the user can specify the type of activation function to implement in the network; choosing one from the following list: logistic sigmoid, linear, or softmax function. The activation functions are shown in the following equations.

$$f(x) = \frac{1}{1 + e^{-x}}$$

Figure 3: Logistic sigmoid activation function.

$$f(x) = x$$

Figure 4: Linear activation function.

$$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}} \text{ for } i = 1, \dots, J$$

Figure 5: Softmax activation function.

The softmax is dissimilar to the other two functions, in that it is not a function of a single fold from the previous layer(s). Essentially, the activation function maps the neuron input into a desired, predetermined range. The ranges for the functions offered in the `create ANN()` Python class are (0,1), $(-\infty, \infty)$, and (0,1), respectively. These function will calculate the unique output for the activated neuron and is used for forward propagating the output through a network. That this case, the created `ANN()` Python class will iterate over each neuron in a hidden layer, calculate the output based on the inputs from the previous layer and the activation function

chosen, and forward propagate the weights through each layer until the output layer is reached. Let's test the activation functions on the previous example network and examine the output.

```
In [20]: network = [
    {'weights': [0.7270700037509241, 0.7383627445080684, 0.18978193074880423]},
    {'weights': [0.7869285898008361, 0.11191242288793657, 0.6323166547815777]},
    {'weights': [0.5542929997136264, 0.6299108552149877, 0.036872224253109986]}
  ],
  {'weights': [0.9541070703271004, 0.9002708967220981, 0.5574211004774138, 0.56239456051497]},
  {'weights': [0.5772579431172555, 0.5744375394342741, 0.276212229019259, 0.5615663370673547]},
  {'weights': [0.49917371980390934, 0.2289715704548373, 0.9165265896818503, 0.4831653543316994]}
],
  {'weights': [0.7936276360327776, 0.9247539681130554, 0.08899479960010304, 0.6728626611317721]},
  {'weights': [0.19170994036771682, 0.8864472297353635, 0.1201225021016884, 0.15257603530797048]}
]
forward_propagate(network,[0,0])

Out[20]: [0.89844089276778516, 0.75519580178017232]
```

Figure 6: Forward Propagation output for example network with a sigmoid activation function.

The default activation function for the created *forward_propagate* Python function is sigmoid, which is shown in the previous Figure 6. The function propagates the input values [0,0] through the network and produces an output array with two values corresponding to the neurons in the output layer. This array will be compared to the known classification values using an error calculation.

To select the appropriate weights at each layer, the error between the forward propagated values and the known values in a training set can be used. This is accomplished by using the first derivative of the activation function to calculate the output slope from a neuron and multiplying the error (i.e. difference between the calculated and known values) by the derivative, or slope. This calculation formula is provided below.

$$\text{Error} = (\text{Calculated Value} - \text{Known Value}) * \text{Derivative}(\text{Known Value})$$

Figure 7: General formula for calculating error in ANN.

The known values correspond to the classification value in the training set. This calculation is relatively straightforward for calculating the error for the output layer neurons but is more convoluted for the hidden layers. The hidden layer error calculation begins with the initial error output layer calculation and iterate through the network layer-by-layer until the input layer is reached, updating the weights along the way. That is, the error is calculated backwards from the output to the input layer across the hidden layers, hence the term back propagation. The error for each neuron in the hidden layers is calculated using the same method provided in Figure 3 via the product of the error and slope (i.e. derivative of the chosen activation function). The following pseudocode (Python) illustrates the general method of implementation, where the errors are calculated by iterating through the forward propagated network in reverse order.

```

1 for i in range(0, len(network))[::-1]:
2     current_layer = network[i]
3     error_list = []
4     errors = [e1-e2 for e1,e2 in zip(current_layer, current_layer_known)]
5     for j in range(len(current_layer)):
6         error = float(0)
7         neuron = current_layer[j]
8         neuron['error'] = errors[j] * derivative(current_layer_known[j])

```

Figure 8: Pseudocode for back propagation.

The Python class uses a dictionary to hold the calculated errors between the expected and known values. The dictionary can then be called layer-by-layer to back propagate the error for each neuron in each hidden layer. The loop in the pseudocode will continue for each hidden layer until the input layer is reached.

Now that the error has been calculated for the initial network weights, the weights need to be updated accordingly to converge on the correct, known classification. This is accomplished in an online manner, where the weights are updated each iteration.² The weights are changed by an amount specified by the use, called the learning rate. The learning is conventionally a fraction (i.e. between 0 and 1) and controls the amount of change in the weights to correct for the calculated error. If a smaller value is chosen for the learning rate, a larger number of training iterations or epochs may be required to obtain a well-trained, low error network and avoids fast convergence of weights to minimize the error (i.e. premature convergence). The weights in the `create ANN()` class are updated using the equation in Figure 9, where w is a given weight for the i^{th} iteration or epoch, LR is the predefined learning weight, E represents the error calculated from the back propagation, and I represents the specific input value.

$$w_{i+1} = w_i + (LR * E * I)$$

Figure 9: Formula to update network weights based on error and learning rate.

Ideally, the weights would converge to a value, thus the error would converge to a minimum given many iterations, or epochs. As the errors lower due to the updated weights in the network, the training step ceases when either a predetermined number of epochs are completed, or a set error value is met on the training dataset. Upon completion of the training step, then network is used to predict the classification on the test dataset given the weights created during the training set. This step is rather straightforward and employs the forward propagation method to move the input data from the test set through the trained network to obtain a classification prediction by selecting the class with the largest probability.

3.1 Results

Let's examine relatively simple network using the spam email dataset with a learning rate of 0.3 (or 30%), two hidden layers with 2 neurons each layer, sigmoid activation function, and 60%-40% training-test split of the data. Of note, this is small network considering the number of

² Another form of updating is batch learning which accumulates the errors across each epoch before updating the network weights.

features in the data is 57. Additionally, recall that the spam email data is organized in a manner that contains the classes in grouped together (see Figure 2). The training-test split utilized in the `ANN()` class randomly shuffles the data (i.e. Python function `random.shuffle` from the `random` Python library) to obtain a more mixed sample in the training and test set, respectively. Performing the `ANN()` class on the dataset for 5 epochs, the class provides the squared error calculated between the predicted and known values for each epoch along with the total elapsed time, as provided in Figure 10 from the iPython notebook and the training error each epoch provided in Figure 11.

Running created ANN Class

Training network on dataset with 57 features and 2 outputs. The number of hidden layers and the number of neurons per hidden layer are variable. I start by **NORMALIZING** the dataframe.

```

1 # Normalizing the dataframe
2 df_norm = (df - df.mean()) / (df.max() - df.min())
3
4 # ANN(data,n_inputs,n_hidden,n_neurons,n_outputs,activation_func,learning_rate)
5 X = ANN(df_norm, #data
6         57, # number of inputs
7         2, # number of hidden layers
8         2, # number of neurons per hidden layer
9         2, # number of outputs (2)
10        'sigmoid', # activation function
11        0.3, # learning rate
12        printer=True)
13 total_error,outputs = X.main(60,5)

```

CHECK: Original data shape of (4601,58) and splits shapes:
 training_data: (2761,58)
 test_data: (1840,58)

epoch	error	time	(seconds)
epoch: 1	error: 24.8854	time: 0.7	(seconds)
epoch: 2	error: 0.9806	time: 1.4	(seconds)
epoch: 3	error: 0.5498	time: 2.1	(seconds)
epoch: 4	error: 0.3805	time: 2.8	(seconds)
epoch: 5	error: 0.2906	time: 3.5	(seconds)

Figure 10: Screenshot of iPython Notebook for created `ANN()` class using sigmoid function.

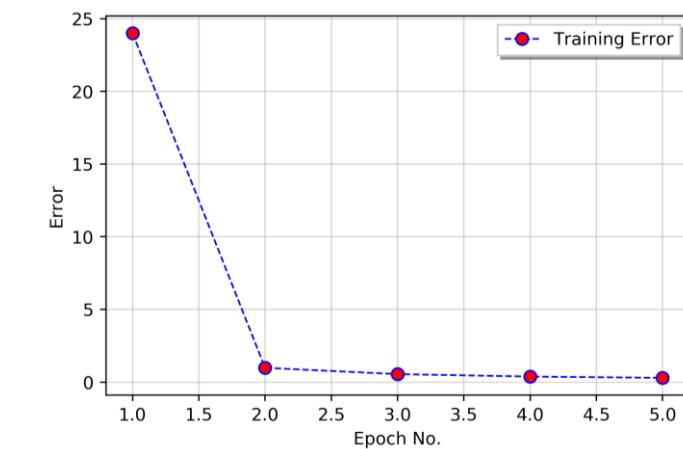


Figure 11: Training error for initial `ANN()` class run with sigmoid function.

The 5-epoch network training is completed in approximately 3.5 seconds with the squared error dropping from roughly 24 to 0.3 using the 2761 length training set. Using the trained network with the updated weights, the `ANN()` class performs a forward propagation step to predict the classification of the test data. The predicted classification accuracy of the network resulted in an accuracy and F1 score of 74.5% and 0.8543.³

To add context to this initial simulation, the activation function is changed to linear with all other network attributes remaining unchanged. The `ANN()` class (screenshot shown in Figure 12 showing implementation of linear function simulation) provided a comparatively lower error of approximately 0.28 in 3.6 seconds for 5 epochs.

Running created ANN Class

Training network on dataset with 57 features and 2 outputs. The number of hidden layers and the number of neurons per hidden layer are variable. I start by **NORMALIZING** the dataframe.

```

1 # Normalizing the dataframe
2 df_norm = (df - df.mean()) / (df.max() - df.min())
3
4 # ANN(data,n_inputs,n_hidden,n_neurons,n_outputs,activation_func,learning_rate)
5 X = ANN(df_norm, #data
6         57, # number of inputs
7         2, # number of hidden layers
8         2, # number of neurons per hidden layer
9         2, # number of outputs (2)
10        'linear', # activation function
11        0.3, # learning rate
12        printer=True)
13 total_error, outputs = X.main(60, 5)

```

CHECK: Original data shape of (4601,58) and splits shapes:
 training_data: (2761,58)
 test_data: (1840,58)

epoch	error	time	(seconds)
epoch: 1	error: 13.7105	time: 0.7	(seconds)
epoch: 2	error: 0.8997	time: 1.4	(seconds)
epoch: 3	error: 0.5147	time: 2.1	(seconds)
epoch: 4	error: 0.3607	time: 2.9	(seconds)
epoch: 5	error: 0.2775	time: 3.6	(seconds)

Figure 12: Screenshot of iPython Notebook for created `ANN()` class using linear function.

³ Accuracy is calculated by $(TN + TP) / (TN + TP + FN + FP)$, where TP = # true positives, TN = # true negatives, FP = # false positives, FN = # false negatives. The F1 score metric is calculated by $F1 \text{ Score} = 2 \times PPV \times TPR / (PPV + TPR)$, where PPV is positive predictive value and TPR is true positive rate.

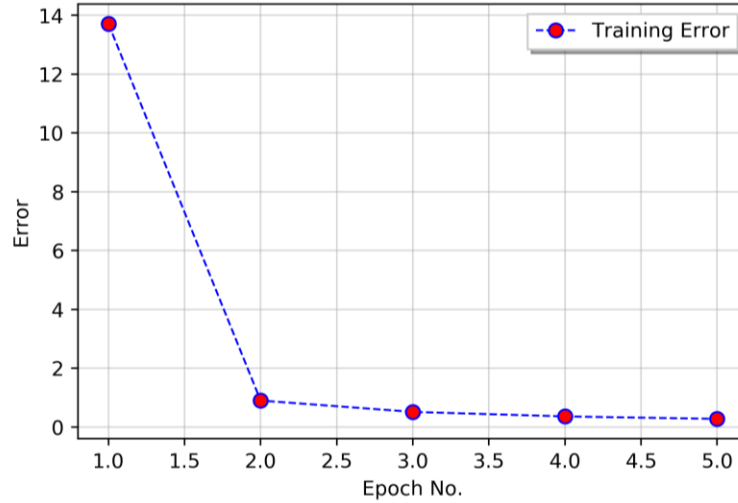


Figure 13: Training error for initial ANN() class run with linear function.

The linear activation function obtained classification accuracy and F1 score of 75.6% and 0.861, respectively. Realistically, 75% accuracy may be adequate for a spam filter, but the accuracy of the model may be improved if the two hyperparameters discussed in the first two questions of section 3 Data Analysis: number of hidden layers and the number of neurons per hidden layer.

Let's begin by adding additional hidden layers to the previously constructed neural networks. Beginning with the sigmoid function, an additional hidden layer with two neurons each was added the neural network in an iterative fashion. The newly composed networks were used to predict the classification on a test set with the model accuracy and F1 score also calculated. The results from this analysis are provided in Tables 2 and 3 where all networks parameters remained equal to the previous networks except the number of hidden layers (i.e. ranging from 2 to 8) and the number of epochs increased to 25.

Table 3: Performance metrics for ANN with multiple hidden layers using sigmoid function.

No. Hidden Layers	Accuracy	F1 Score
2	0.7495	0.8568
3	0.7723	0.8715
4	0.7663	0.8677
5	0.7571	0.8617
6	0.7587	0.8628
7	0.7620	0.8649
8	0.7804	0.8767

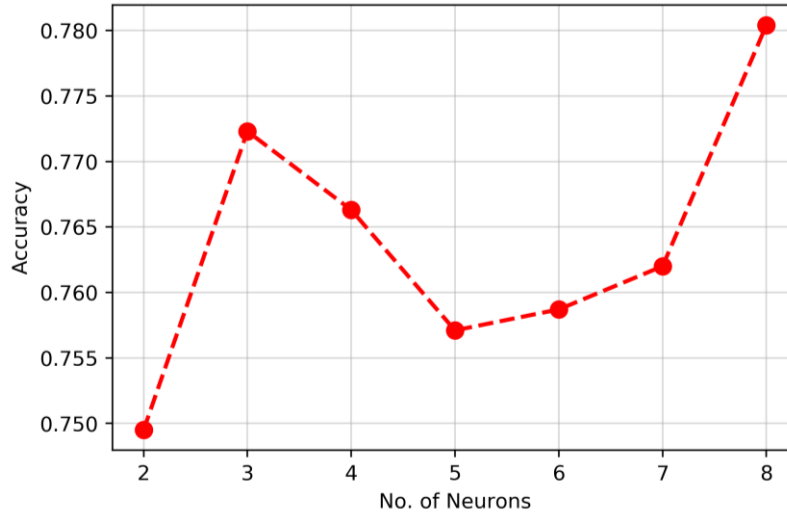


Figure 14: ANN accuracy with sigmoid function for changing number of hidden layers.

Table 4: Performance metrics for ANN with multiple hidden layers using linear function.

No. Hidden Layers	Accuracy	F1 Score
2	0.7522	0.8586
3	0.7668	0.8680
4	0.7641	0.8663
5	0.7609	0.8642
6	0.7598	0.8635
7	0.7636	0.8659
8	0.7663	0.8677

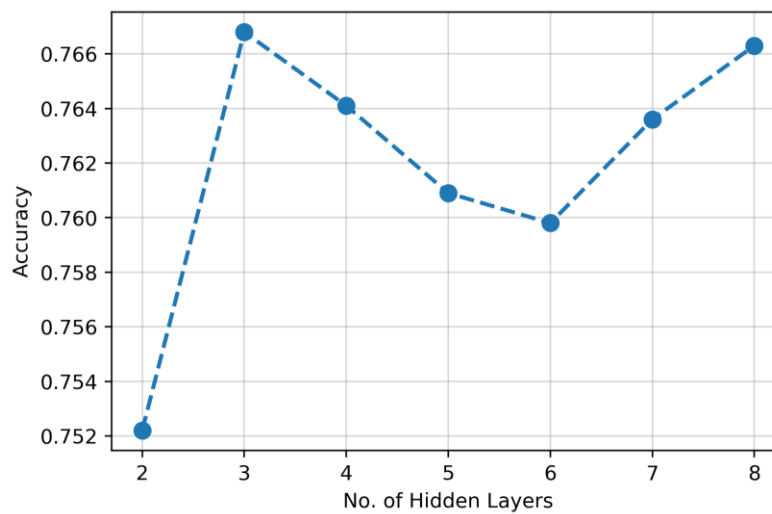


Figure 15: ANN accuracy with linear function for changing number of hidden layers.

From the two previous Tables, the accuracy of the both models do not decrease for each added hidden layer, which is typical for training neural networks. As the complexity of the network increase, the training error is fixed but the validation error begins to increase as the network starts to overfit the data. Furthermore, since the number of epochs were increased from 5 to 25, additional considerations must be considered. That is, a typical behavior of a network when trained too long, the validation error starts to increase, and the network starts to overfit. Similarly, due to the nonlinearity of the network, the error function can have multiple minima. These issues highlight the bias-variance tradeoff that typical accompanies such predictive models.

To further assess the architecture of the network, the neurons per hidden layer are increased from 2 to 30, skipping every other number (i.e. only even numbers from 2 to 30). Using the information from the previous two Tables and to save time as training a neural network can be computationally expensive, the number of hidden layers employed in the network for this analysis is set to three due to the comparatively high level of accuracy for both functions. Of note, this analysis utilizes a different training-test split, thus will not be equal to the previous analysis provided in Tables 2 and 3.

Table 5: Performance metrics for ANN with changing number of neuron using sigmoid function.

No. of Neurons	Accuracy	F1 Score
2	0.7690	0.8694
4	0.7929	0.8845
6	0.7446	0.8536
8	0.7701	0.8701
10	0.7527	0.8589
12	0.7625	0.8652
14	0.7587	0.8628
16	0.7663	0.8677
18	0.7495	0.8568
20	0.7446	0.8536
22	0.7625	0.8652
24	0.7603	0.8638
26	0.7576	0.8621
28	0.7652	0.8670
30	0.7505	0.8575

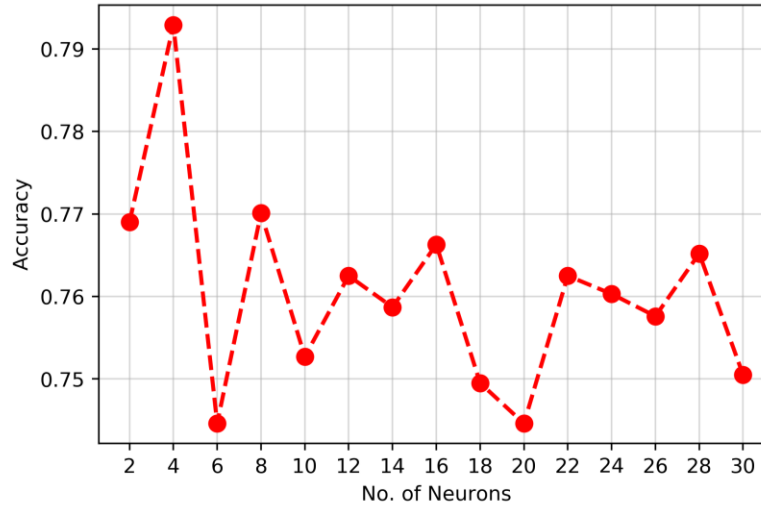


Figure 16: ANN accuracy using sigmoid function with changing number of neurons on three hidden layers.

Table 6: Performance metrics for ANN with changing number of neuron using linear function.

No. of Neurons	Accuracy	F1 Score
2	0.7609	0.8642
4	0.7886	0.8796
6	0.7712	0.8708
8	0.7527	0.8589
10	0.7565	0.8614
12	0.7397	0.8504
14	0.7630	0.8656
16	0.7739	0.8725
18	0.7636	0.8659
20	0.7674	0.8684
22	0.7690	0.8694
24	0.7386	0.8496
26	0.7576	0.8621
28	0.7663	0.8677
30	0.7457	0.8543

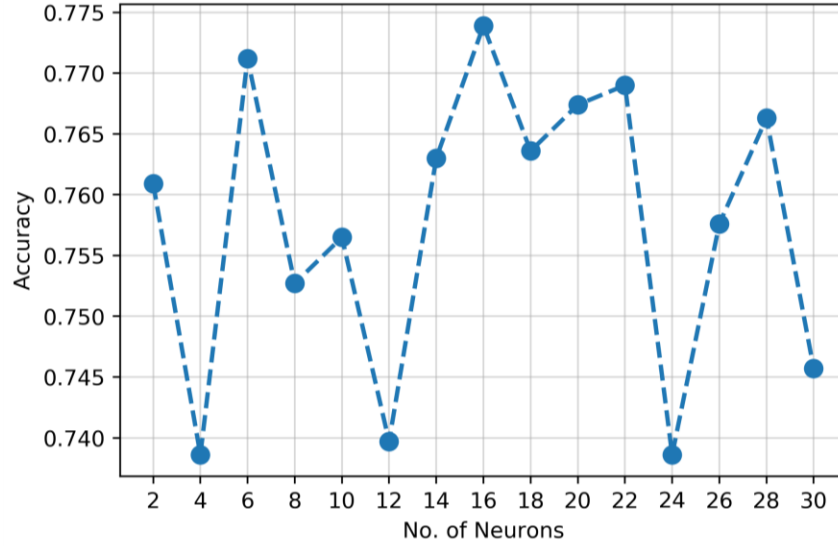


Figure 17: ANN accuracy using linear function with changing number of neurons on three hidden layers.

From the above analysis to determine the number of neurons per hidden layer, it is difficult to choose as the accuracy of the model is not strongly correlated to the number of neurons employed; which is perhaps incorrect or inaccurate. In general, the least number of hidden neurons to meet a level of accuracy should be picked to lower the number of weights to be calculated. Let's use the data analysis and choose the optimal number of neurons based on the highest accuracy while considering the number of neurons. This would result in 4 hidden neurons per layer for the ANN employing the sigmoid activation function and 6 hidden neurons per layer for the ANN employing the linear activation function.

3.2 Dimensionality Reduction Analysis

Recall that during the data exploration steps, several attributes were observed, such as 37 features with equal to or greater than 80% zero-valued entries with 8 features containing more than 95% zero-valued entries. A dimensionality reduction technique, such as principal component analysis or PCA, can be utilized to decrease the number of features used in the ANN prediction model, while maintaining a level of total variance explained. PCA was used on the dataset using the Python library *scikit-learn* to reduce the dimensionality via singular value decomposition. The variance explained and the cumulative (total) variance explained by each principal component is shown in Figure 18 after z-standardizing and mean-centering the data.

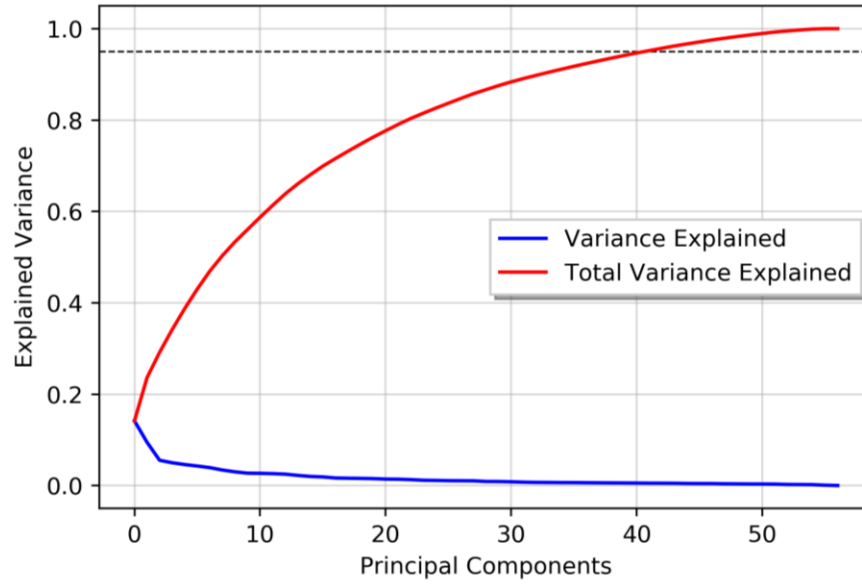


Figure 18: Variance explained for each singular value using PCA.

The horizontal (black) dashed line represents 95% of total variance explained, which is the value utilized in this project to lower the dimensionality of the data. The 41th principal component, or singular value, barely surpasses the 95% of total variance explained and thus, the first 41 components will be used in this analysis. Employing the *PCA.transform()* function from the *sklearn.decomposition* Python library fits the model and lower the dimensionality to 41, which was set in the original object call via *n_components* parameter (i.e. *PCA(n_components)*). The created *ANN()* class was performed on the lower dimensional dataset using both the ANN with sigmoid and linear activation functions' optimal parameters; 3 hidden layers with 4 hidden neurons and 3 hidden layers with 6 hidden neurons, respectively. The ANN using sigmoid activation function resulted in a 75.9% accuracy and F1 score of 0.8628, while the ANN using a linear activation function resulted in a 70.2% accuracy and F1 score of 0.825 (see Figure 19 and Figure 20).

```

1 # ANN(data,n_inputs,n_hidden,n_neurons,n_outputs,activation_func,learning_rate)
2 X = ANN(lower_dim_data, #data
3         41, # number of inputs
4         3, # number of hidden layers
5         4, # number of neurons per hidden layer
6         2, # number of outputs (2)
7         'sigmoid', # activation function
8         0.3, # learning rate
9         printer=False # prints error per epoch (used during diagnostic testing)
10        )
11 number_of_epochs = 25 # number of epochs
12 split_in_training = 60 # % of data in training set
13 total_error, outputs = X.main(split_in_training, number_of_epochs)
14
15 TN, FN, TP, FP = predict_accuracy(X.data_test, outputs, prints=False)
16 Acc, TPR, PPV, TNR, F1 = getMetrics(TN, FN, TP, FP)

```

```

===== Performance Metrics =====
Acc. : 0.7587
F1 : 0.8628
=====

```

Figure 19: Screenshot of PCA data for ANN with sigmoid function.

```

1 # ANN(data,n_inputs,n_hidden,n_neurons,n_outputs,activation_func,learning_rate)
2 X = ANN(lower_dim_data, #data
3         41, # number of inputs
4         3, # number of hidden layers
5         6, # number of neurons per hidden layer
6         2, # number of outputs (2)
7         'linear', # activation function
8         0.3, # learning rate
9         printer=False # prints error per epoch (used during diagnostic testing)
10        )
11 number_of_epochs = 25 # number of epochs
12 split_in_training = 60 # % of data in training set
13 total_error, outputs = X.main(split_in_training, number_of_epochs)
14 TN, FN, TP, FP = predict_accuracy(X.data_test, outputs, prints=False)
15 Acc, TPR, PPV, TNR, F1 = getMetrics(TN, FN, TP, FP)

```

```

===== Performance Metrics =====
Acc. : 0.7022
F1 : 0.8250
=====

```

Figure 20: Screenshot of PCA data for ANN with linear function.

The dimensionality reduced data still maintained an adequate level of accuracy compared to the original, full data set using all 57 features as the input neurons on the input layer.

References

Alpaydin, Ethem. *Introduction to Machine Learning*. Third Edition. The MIT Press. Cambridge, Massachusetts. (2014). ISBN: 978-0-262-02818-9.