Neural Networks for Spam Email Detection

CS 4701 Practicum in Artificial Intelligence

Chris Anderson (cma227) and Alex Luo (hl532)

December 12, 2013

1 Introduction

We implemented feedforward neural networks with varying numbers of neurons and layers, trained by the standard back-propagation algorithm. Using an available dataset of emails from the UC Irvine Machine Learning Repository, we trained the neural nets to detect spam emails, varying the parameters of the neural nets.

2 Spam detection

Spam email detection and filtering is one of the classic problems in machine learning. A large proportion of emails sent are spam containing advertisements, scams, and other things that the average email user would much rather not see in their inbox. To combat this, email providers implement spam filters, which follow the general principles of machine learning: given emails that are known to be spam or not spam, or that have been identified as one or the other by the user, use the information provided by the text in those emails to learn rules that can classify future emails as spam or not spam.

The most common way of representing textual data for machine learning is what is known as a "bag of words" representation. One determines words that are likely to

appear in examples of each class, and processes the documents in the data set to reduce each one to an array of how many times each of those chosen keywords appears. Since this is typically a time-consuming process, both in terms of figuring out which words make good attributes and in terms of writing code to perform that processing, we looked for data which was already available in the bag-of-words format.

The UC Irvine Machine Learning Repository has such a dataset, the "Spambase". This widely used dataset contains 4601 emails collected by employees at HP Labs in 1999, of which 1813 are spam. The data is provided in a comma-separated-value file, with each email reduced to 57 numerical attributes and a label of 0 for not spam or 1 for spam. Therefore, machine learning on the emails entails finding a function from those 57 inputs to a binary label. The attributes are 48 values of word frequency (percentage of words in the email that are the given word), 6 values of punctuation/character frequency (percentage of characters that are that character), and 3 values describing the usage of caps lock as follows:

1. make	10. mail	19. you	28. 650
2. address	11. receive	20. credit	29. lab
3. all	12. will	21. your	30. labs
4. 3d	13. people	22. font	31. telnet
5. our	14. report	23. 000	32. 857
6. over	15. addresses	24. money	33. data
7. remove	16. free	25. hp	34. 415
8. internet	17. business	26. hpl	35. 85
9. order	18. email	27. george	36. technology

37. 1999
38. parts
39. pm
40. direct
41. cs
42. meeting
43. original
44. project
45. re
46. edu
47. table
48. conference
49. semicolon
50. parentheses
51. square brackets
52. exclamation point
53. dollar sign
54. pound sign
55. average consecutive capital letters
56. largest consecutive caps
57. total number of caps

Overall, it is pretty clear from looking at the word frequency features which are clearly intended to indicate spam (credit, free) and which are clearly generic workplace emails used as non-spam (hp, labs, conference). The use of features specific to the workplace where the data was collected would limit the ability of any spam filter trained on this data to generalize to other data, but since we only intend to show that we can build a spam filter that is not a major concern.

In practice, spam detection is typically done with statistical methods like the Bayes classifier, which uses the data to determine the probability of each word appearing in a spam email, and then uses those probabilities to determine the probability an email is spam given all the words that appear in it. That method typically works well on text; however, it is more in the spirit of AI (and what we have covered) to use a neural network and see if it has reasonable performance.

The spambase data was divided at random into two data sets. 3000 examples were used for training the neural nets, while the remaining 1601 were used as test data to evaluate the performance of the resulting networks.

3 Neural networks

Artificial neural networks are a class of machine learning algorithms in which the common feature is that a function is computed by a set of "neurons". These mathematical objects, which are assumed to work somewhat similarly to neurons in the brain, take a set of inputs and output the result of their activation function on a weighted sum of inputs: given activation function g, weights $w_{0,j}...w_{n,j}$, and inputs $a_0...a_n$, neuron j outputs:

$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j}a_i)$

g is typically either a hard threshold function which returns 1 if in_j is above a certain value and 0 if it is below, or a logistic/sigmoid function, sigmoid(x) $= 1/(1+e^{-x})$ which scales slowly from 0 to 1 over a range, but has the advantage of being differentiable which wewill need later.

We concerned ourselves with feed-forward neural networks, as described in the text; networks in which a neuron will never have its output go to a previous neuron. In feed-forward networks, there is an input layer, some number of hidden layers, and an output layer, connected such that in every layer, all neurons receive the activations of all neurons in the previous layer, and if applicable output their activations to all neurons in the next layer.

A feed-forward neural network with zero hidden layers is known as a perceptron, and is a simplified version where all inputs connect directly to the output. This means that the output is simply a linear combination of the inputs, which on the one hand makes perceptrons and their outputs easier to understand, but on the other hand means that if the data is not linearly separable a perceptron will never be able to get perfect accuracy on it. Despite that shortcoming, perceptrons are widely used because the majority of real-life data will either be linearly separable or almost linearly separable to the point where a perceptron still does well, and because they take comparatively less time to train.

Because they are effectively perceptrons stacked on top of each other, feed-forward neural networks with one or more hidden layers are often referred to as multilayer perceptrons. They are more difficult to train, but gain the advantage of being able to exactly learn any arbitrary function given sufficient time and training examples.

The algorithm to "feed forward" inputs to compute an output is as follows:

```
for layer in (first hidden layer, ..., output layer)
  for each neuron in the layer
    weighted sum previous layer's values
    take activation function of sum
return the activation values of the output layer
```

Training is done by the standard back-propagation algorithm: the network computes its output for a given input with the current weight values, then compares that to the output values that we know are correct to compute the "delta" of how much that output is off by. Then, since theoretically each node in the previous layer is responsible for an amount of that error proportional to its weight, we create the back-propagation rule:

$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$

and use that to find delta for every node working backwards to the first hidden layer. (g' is the derivative of the activation function, in_j is the sum from feeding forward.)

Once we have back-propagated the deltas, we use them to update the weights between all neurons in the network by the weight-update rule:

$w_{i,j} = w_{i,j} + \alpha \times a_i \times \Delta_j$

where alpha is an extra "learning rate" parameter that determines how big of a change in the weights is made. Therefore, the back-propagation algorithm to update the weights based on a single training example is:

```
feed forward the training example
for each output node
```

```
  calculate delta
for each layer working backwards to the first
  for each neuron
    use back-propagate rule to get delta
for each neuron in the entire network
  use weight-update rule to adjust weight
```

So that handles the training for a single example, and we can use it to train a network for all examples. Start out with all weights in the network set to random small numbers. Then, we loop through the data repeatedly until some condition is met (generally a certain number of iterations), back-propagating the example we are on at every iteration.

4 Design

After initial difficulties implementing a generalized neural network that would work with any number of hidden layers, we opted to implement separate programs for the perceptron, neural network with 1 hidden layer, and neural network with 2 hidden layers. This was so that we could guarantee that there would be a working neural net for us to use. Coding was done in Python, using the external numpy library for our arrays, as well as the standard pickle and csv modules to get our data from the csv file.

As previously mentioned, we worked with feedforward neural networks, following essentially the algorithms from the relevant section of the text. The design of the neural network with 1 hidden layer will be used as the representative example for illustrating how our code for the networks was designed.

We created a neural network class, from which one could instantiate new networks with a given number of inputs, outputs, and neurons in the hidden layer. Since there is no state in a feedforward neural network other than the weights, the only instance variable

besides those was the weights, which we represented by storing the weights for a layer in an array, and keeping a list of those. For example, weights[1][2,3] would be the weight between the second neuron in the 1st (hidden, counting inputs as 0th) layer and the 3rd neuron in the 2nd (output) layer. Once one has instantiated a neural network object, one can call feed_forward(inputs) or back_propagate(inputs, outputs, learning_rate) on it, which both behave exactly as expected according to the given algorithms.

Two additional functions are provided for convenience, train_network which takes lists of all inputs and outputs and a number of iterations and repeatedly back-propagates, and get_output, which feeds an input forward and then returns only the output values as opposed to an array of all of the activations (which is returned because back_propagate needs all of it).

Two other main files were coded as part of this project. The file get_data.py performed the surprisingly annoying task of reading out the comma-separated-value files provided into Python arrays and used Python's built-in pickle module to send them to files that Python could work with more easily. The file test.py, which came in distinct flavors for each of the three neural networks, loaded the attributes and data from the pickle files, trained a neural net, and then called get_output with every example in the data and computed our test accuracy and training accuracy.

Before using the neural net classes on the actual data, they were run with simple Boolean functions to make sure that they worked properly. As an example of this, a neural network with one hidden layer of 3 neurons trained on XOR for 5000 iterations (4 possible examples, so 20000 updates to the weights) returned (1,1) $\rightarrow$ 0.04765889, (1,0) $\rightarrow$ 0.96447403, (0,1) $\rightarrow$ 0.96025087, and (0,0) $\rightarrow$ 0.02194868. Both the examples that should be labeled 1 returned 0.96, and both those that should be 0 returned less than 0.05, so because the sigmoid function does not actually return integers and only gets infinitely close to 1, getting labels close to 0 and 1 indicates a correct result (because of this, when computing the test accuracy and training accuracy of examples I had it

round the output of the neural net to 0 or 1).

The neural network with 2 hidden layers differed only in the inclusion of a second parameter when creating the network, to set the number of neurons in the second hidden layer, and performed similarly well on the XOR test.

Single layer perceptrons are incapable of learning XOR because it is not linearly separable, but it worked admirably on learning AND and OR, indicating that that function was working properly as well.

5 Testing

Once we had ascertained that the neural networks were working properly, we attempted to use them on the Spambase email data, training on the training set and then reporting the accuracy on the training and test sets. Initially, this was unsuccessful:

Perceptron	.599 training accuracy	.616 test accuracy
1 hidden layer	.401 training accuracy	.381 test accuracy
1 hidden layer	.401 training accuracy	.381 test accuracy

The problem with those values, which occurred regardless of any parameters, is that 1813/4601, or .394 of the overall data set is spam, so 60% accuracy can be achieved by guessing there is no spam, and 40% accuracy is guessing that every email is spam. Not only are both of those inaccurate, they defeat the purpose of a spam filter by not making a serious attempt to distinguish spam and non-spam.

We examined the data to try to see why neural nets would not converge, and noticed that, according to the file documenting the spambase data, every attribute had an average value of less than 1 except for the three attributes about the email's usage of caps lock, which were much higher - in particular, the attribute for the total number of capital letters typically had a value in the hundreds. We reasoned that, not only were those features not a "bag of words" feature, but that having features that were consistently orders of magnitude larger than the others would likely make it so that the

delta on those features was larger than other deltas, giving them undue influence when updating weights in back-propagation. To see if this was the case, we modified the script get_data to throw out the three "caps lock" attributes.

The results of this were even more effective than we had hoped, almost immediately giving us 90% accuracy across the board using the data with only bag-of-words features.

For perceptrons:

Iterations	Training accuracy	Test accuracy
1	.922	.920
10	.933	.923
50	.930	.911
100	.939	.926

Doing more iterations generally increased training accuracy, while not having much effect on test accuracy. This shows that more iterations primarily causes the perceptron to overfit the training set without generalizing better to the test set by the same amount, though its test accuracy did still increase. Note that one "iteration" is one run through back-propagating the entire training set, so each iteration is 3000 weight updates.

For neural networks with a single hidden layer:

Iterations	Hidden neurons	Training accuracy	Test accuracy
1	10	.904	.890
5	10	.925	.911
10	10	.931	.912
1	20	.907	.902
5	20	.928	.923
10	20	.926	.906
1	40	.401	.381
5	40	.401	.381

There seemed to be a "sweet spot" around 5 iterations and 20 hidden neurons that gave us the best training and test accuracy. However, at all other points it performed worse than the perceptron. Also, 40 neurons did not converge at all in the number of iterations that we were willing to wait for it to run for, and had the same performance as we were getting before removing the caps lock features from the data.

Lastly, neural networks with 2 hidden layers took a lot more time to train without a significant gain in accuracy at detecting spam. (Runs with 1 iteration were omitted because in all cases they did not converge during that iteration.)

Iterations	1st hidden layer	2nd hidden layer	Training accuracy	Test accuracy
5	5	5	.919	.911
5	10	5	.927	.919
5	20	10	.599	.619
10	20	10	.929	.916

Again, though both 5-10-5 and 10-20-10 performed reasonably well, they were beaten by both the best single layer network and best perceptron.

6 Results and conclusions

This work demonstrated that a feedforward neural network can be an effective means of detecting spam emails with over 90% accuracy, but also that there are no appreciable benefits for using a multi-layer network instead of using a simple perceptron.

While this is very accurate, perhaps more so than we expected, it still does not make neural network spam filtering something that one would want to use in practice necessarily. For example, most users would much rather have the occasional spam email end up in their inbox than to have an overzealous spam filter that detects all spam but also gets false positives on important emails. 90% accuracy where the other 10% are spam that get through the filter would be a mild nuisance, whereas 90% accuracy where most of the other 10% are emails we wanted to read that ended up in the spam folder would be actively undesirable. So one future extension would be to measure the number of false positives given by the neural network, and if possible try to minimize them.

Also, our 90-92% accuracy is still vastly outperformed by Bayesian spam filtering, which as we mentioned at the start of the paper is the standard algorithm used in practice for spam filtering. With a Bayesian classifier or other probabilistic/statistical models, it is possible to achieve accuracy greater than 99% and barely any false positives.

Lastly, as previously mentioned the keywords used as attributes in the Spambase data set are fairly specific to the workplace where they were collected, not to mention that the data was collected over 10 years ago. It is likely that many spammers have adapted to the presence of spam filters and tried to design emails that will not be caught, for example by including unrelated and innocuous text that would probably contain words associated with non-spam emails. For this reason, it would be interesting to see whether neural networks, and spam filters in general, perform significantly better or worse on email data sets collected more recently and from more "generic" emails, and how well neural network-based spam filters can generalize to new emails on such data.

So while a neural network is therefore unlikely to see serious use in practice as a means of spam detection simply because it will always be outclassed by other methods, once we removed the problematic features from the data our neural networks had significant performance that demonstrates they were learning successfully.

7 Sources

Russell and Norvig, "Artificial Intelligence: A Modern Approach". Pearson, 2010.

George Forman, "Spambase Data Set". UC Irvine Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets/Spambase

William S. Yerazunis. "The spam filtering plateau at 99.9% and how to get past it". 2004. http://crm114.sourceforge.net/docs/Plateau99.pdf

CS 4700 lecture slides

Python libraries used: numpy

8 Appendix: code

8.1 perceptron.py

```
import numpy as np

# Sigmoid function for activation
def sigmoid(x):
    return 1/(1 + np.exp(-x))

# Derivative of sigmoid for back-propagating
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

class Perceptron:

    def __init__(self, num_inputs, num_outputs):
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs

        self.weights = np.random.rand(self.num_inputs + 1, self
            .num_outputs)

    def feed_forward(self, inputs):
        inputs = np.append(inputs, 1.) # bias
        sums = np.random.rand(self.num_outputs)
        activations = np.random.rand(self.num_outputs)

        for output in range(self.num_outputs):
            sum = 0
            for input in range(self.num_inputs + 1):
                sum += self.weights[input, output] * inputs[
                    input]
            sums[output] = sum
            activations[output] = sigmoid(sum)

        return (sums, activations)

    def back_propagate(self, inputs, given_outputs,
        learning_rate):
        (sums, activations) = self.feed_forward(inputs)
        inputs = np.append(inputs, 1.)
        deltas = np.random.rand(self.num_outputs)

        for output in range(self.num_outputs):
            deltas[output] = sigmoid_derivative(sums[output]) *
                (given_outputs[output] - activations[output])
```

```
        for input in range(self.num_inputs + 1):
            for output in range(self.num_outputs):
                self.weights[input, output] += (learning_rate *
                    inputs[input] * deltas[output])

    def train_network(self, num_examples, inputs_list,
        outputs_list, num_iterations, learning_rate):
        for i in range(num_iterations):
            for j in range(num_examples):
                self.back_propagate(inputs_list[j],
                    outputs_list[j], learning_rate)

    def get_output(self, inputs):
        return self.feed_forward(inputs)[1]
```

8.2 neural_network_1_hidden.py

```
class NeuralNetwork_1HL:

    def __init__(self, num_inputs, num_hidden_neurons,
        num_outputs):
        self.num_inputs = num_inputs
        self.num_hidden_neurons = num_hidden_neurons
        self.num_outputs = num_outputs

        # +1 for bias
        self.weights = [np.random.rand(self.num_inputs + 1,
            self.num_hidden_neurons), np.random.rand(self.
            num_hidden_neurons, self.num_outputs)]

    def feed_forward(self, inputs):
        # add 1 to end of inputs for bias
        inputs = np.append(inputs, 1.)

        sums = [np.random.rand(self.num_hidden_neurons), np.
            random.rand(self.num_outputs)]
        activations = [np.random.rand(self.num_hidden_neurons),
            np.random.rand(self.num_outputs)]

        for neuron in range(self.num_hidden_neurons):
            sum = 0
            for input in range(self.num_inputs + 1):
                sum += (self.weights[0][input, neuron] * inputs
```

```
                [input])
            sums[0][neuron] = sum
            activations[0][neuron] = sigmoid(sum)

        for output in range(self.num_outputs):
            sum = 0
            for neuron in range(self.num_hidden_neurons):
                sum += (self.weights[1][neuron, output] *
                    activations[0][neuron])
            sums[1][output] = sum
            activations[1][output] = sigmoid(sum)

        return (sums, activations)

    def back_propagate(self, inputs, given_outputs,
        learning_rate):
        (sums, activations) = self.feed_forward(inputs)
        inputs = np.append(inputs, 1.)

        deltas = [np.random.rand(self.num_hidden_neurons), np.
            random.rand(self.num_outputs)]

        # back propagate deltas
        for output in range(self.num_outputs):
            deltas[1][output] = sigmoid_derivative(sums[1][
                output]) * (given_outputs[output] - activations
                [1][output])

        for neuron in range(self.num_hidden_neurons):
            sum = 0
            for output in range(self.num_outputs):
                sum += (self.weights[1][neuron, output] *
                    deltas[1][output])
            deltas[0][neuron] = (sigmoid_derivative(sums[0][
                neuron]) * sum)

        # use deltas to update weights
        for input in range(self.num_inputs + 1):
            for neuron in range(self.num_hidden_neurons):
                self.weights[0][input, neuron] += (
                    learning_rate * inputs[input] * deltas[0][
                    neuron])
```

```
            for neuron in range(self.num_hidden_neurons):
                for output in range(self.num_outputs):
                    self.weights[1][neuron, output] += (
                        learning_rate * activations[0][neuron] *
                        deltas[1][output])


    # Train a network by repeatedly backpropagating all the
        examples
    def train_network(self, num_examples, inputs_list,
        outputs_list, num_iterations, learning_rate):
        for i in range(num_iterations):
            for j in range(num_examples):
                self.back_propagate(inputs_list[j],
                    outputs_list[j], learning_rate)

    def get_output(self, inputs):
        return self.feed_forward(inputs)[1][1]
```

8.3 neural_network_2_hidden.py

```
class NeuralNetwork_2HL:

    def __init__(self, num_inputs, num_hidden_first,
        num_hidden_second, num_outputs):
        self.num_inputs = num_inputs
        self.num_hidden_first = num_hidden_first
        self.num_hidden_second = num_hidden_second
        self.num_outputs = num_outputs

        # +1 for bias
        self.weights = [np.random.rand(self.num_inputs + 1,
            self.num_hidden_first), np.random.rand(self.
            num_hidden_first, self.num_hidden_second), np.random
            .rand(self.num_hidden_second, self.num_outputs)]

    def feed_forward(self, inputs):
        # add 1 to end of inputs for bias
        inputs = np.append(inputs, 1.)

        sums = [np.random.rand(self.num_hidden_first), np.
            random.rand(self.num_hidden_second), np.random.rand(
            self.num_outputs)]
        activations = [np.random.rand(self.num_hidden_first),
```

```
            np.random.rand(self.num_hidden_second), np.random.
            rand(self.num_outputs)]

        for neuron in range(self.num_hidden_first):
            sum = 0
            for input in range(self.num_inputs + 1):
                sum += (self.weights[0][input, neuron] * inputs
                    [input])
            sums[0][neuron] = sum
            activations[0][neuron] = sigmoid(sum)

        for next_neuron in range(self.num_hidden_second):
            sum = 0
            for prev_neuron in range(self.num_hidden_second):
                sum += (self.weights[1][prev_neuron,
                    next_neuron] * activations[0][prev_neuron])
            sums[1][next_neuron] = sum
            activations[1][next_neuron] = sigmoid(sum)

        for output in range(self.num_outputs):
            sum = 0
            for neuron in range(self.num_hidden_second):
                sum += (self.weights[2][neuron, output] *
                    activations[1][neuron])
            sums[2][output] = sum
            activations[2][output] = sigmoid(sum)

        return (sums, activations)

    def back_propagate(self, inputs, given_outputs,
        learning_rate):
        (sums, activations) = self.feed_forward(inputs)
        inputs = np.append(inputs, 1.)

        deltas = [np.random.rand(self.num_hidden_first), np.
            random.rand(self.num_hidden_second), np.random.rand(
            self.num_outputs)]

        # back propagate deltas
        for output in range(self.num_outputs):
            deltas[2][output] = sigmoid_derivative(sums[2][
                output]) * (given_outputs[output] - activations
                [2][output])
```

```
        for neuron in range(self.num_hidden_second):
            sum = 0
            for output in range(self.num_outputs):
                sum += (self.weights[2][neuron, output] *
                    deltas[2][output])
            deltas[1][neuron] = (sigmoid_derivative(sums[1][
                neuron]) * sum)

        for prev_neuron in range(self.num_hidden_first):
            sum = 0
            for next_neuron in range(self.num_hidden_second):
                sum += (self.weights[1][prev_neuron,
                    next_neuron] * deltas[1][next_neuron])
            deltas[0][prev_neuron] = (sigmoid_derivative(sums
                [0][prev_neuron]) * sum)

        # use deltas to update weights
        for input in range(self.num_inputs + 1):
            for neuron in range(self.num_hidden_first):
                self.weights[0][input, neuron] += (
                    learning_rate * inputs[input] * deltas[0][
                    neuron])

        for prev_neuron in range(self.num_hidden_first):
            for next_neuron in range(self.num_hidden_second):
                self.weights[1][prev_neuron, next_neuron] += (
                    learning_rate * activations[0][prev_neuron]
                    * deltas[1][next_neuron])

        for neuron in range(self.num_hidden_second):
            for output in range(self.num_outputs):
                self.weights[2][neuron, output] += (
                    learning_rate * activations[1][neuron] *
                    deltas[2][output])


    # Train a network by repeatedly backpropagating all the
        examples
    def train_network(self, num_examples, inputs_list,
        outputs_list, num_iterations, learning_rate):
        for i in range(num_iterations):
            for j in range(num_examples):
```

```
                self.back_propagate(inputs_list[j],
                    outputs_list[j], learning_rate)

    def get_output(self, inputs):
        return self.feed_forward(inputs)[1][2]
```

8.4 get_data_reduced.py - reads csv file and gives us data python can use, throws out the 3 capslock attributes

```
import pickle
import csv

filename = 'spambase.test'
f = open(filename, 'r')
data = []
freader = csv.reader(f, delimiter=',')
for row in freader:
    data.append(row)
f.close()

# There are 58 columns in the data...
# 57 are attributes
# 48 are frequency of a word
# 6 are frequency of a punctuation mark
# 3 are data on sequences of characters
# Last is label
num_attributes = 57

attributes = []
labels = []
for example in data:
    #attributes.append(example[:num_attributes])
    attributes.append(example[:54])
    labels.append(example[num_attributes])

# convert the strings to numbers
for row in range(len(attributes)):
    for col in range(54):
        attributes[row][col] = float(attributes[row][col])

for x in range(len(labels)):
    labels[x] = float(labels[x])

attr_out = open('test_data_reduced.pickle', 'w')
```

```
label_out = open('test_label.pickle', 'w')
pickle.dump(attributes, attr_out)
pickle.dump(labels, label_out)
attr_out.close()
label_out.close()
```

8.5 test.py - only perceptron version shown but all 3 are the same

```
# CHANGE THESE PARAMETERS, OBSERVE RESULTS
num_iterations = 1
learning_rate = 1

print('number of iterations = '+str(num_iterations))
print('learning rate = '+str(learning_rate))
import pickle

import numpy as np

import perceptron
# Sigmoid function for activation
def sigmoid(x):
    return 1/(1 + np.exp(-x))

# Derivative of sigmoid for back-propagating
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

f1 = open('train_data_reduced.pickle', 'r')
f2 = open('train_label.pickle', 'r')
f3 = open('test_data_reduced.pickle', 'r')
f4 = open('test_label.pickle', 'r')
train_attr = pickle.load(f1)
train_label = pickle.load(f2)
test_attr = pickle.load(f3)
test_label = pickle.load(f4)
f1.close()
f2.close()
f3.close()
f4.close()

#num_attrs = 57 # not reduced
num_attrs = 54  # reduced to only frequencies
train_data_size = 3000
test_data_size = 1601
```

```
# all entries in the lists have to be in numpy format
for i in range(train_data_size):
    train_attr[i] = np.array(train_attr[i])
    train_label[i] = np.array([train_label[i]])
for i in range(test_data_size):
    test_attr[i] = np.array(test_attr[i])
    test_label[i] = np.array([test_label[i]])

print('Loaded data')
print('Training perceptron')
per = perceptron.Perceptron(num_attrs, 1)
per.train_network(train_data_size, train_attr, train_label,
   num_iterations, learning_rate)
print('Trained perceptron')

# check training accuracy
num_train_correct = 0
for i in range(train_data_size):
    if per.get_output(train_attr[i])[0] > 0.5:
        if train_label[i][0] == 1:
            num_train_correct += 1
    else:
        if train_label[i][0] == 0:
            num_train_correct += 1

train_accuracy = num_train_correct/float(train_data_size)
print(str(train_accuracy)+' training accuracy for perceptron')
print(str(num_train_correct)+ ' of ' + str(train_data_size))

# check test accuracy
num_test_correct = 0
for i in range(test_data_size):
    if per.get_output(test_attr[i])[0] > 0.5:
        if test_label[i][0] == 1:
            num_test_correct += 1
    else:
        if test_label[i][0] == 0:
            num_test_correct += 1

test_accuracy = num_test_correct/float(test_data_size)
print(str(test_accuracy)+' test accuracy for perceptron')
print(str(num_test_correct)+ ' of ' + str(test_data_size))
```