Vitaliy Shydlonok

CSC 365 Project 3

Report

Data Repositories:

- AFINN (https://github.com/fnielsen/afinn/blob/master/afinn/data/AFINN-en-165.txt)

This document contains 3,382 English phrases that are labeled with a positive or negative semantic value.

- Large Movie Review Dataset (http://ai.stanford.edu/~amaas/data/sentiment/)

This dataset contains 25,000 movie reviews for training and 25,000 reviews for testing. The reviews are split into separate files, each labeled with a positive or negative value between 1 and 10.

Implementation:

I started with an implementation of FeatureVector and a Tfidf calculator from project 2. I used these parts to convert all the training and testing reviews into a format that could be fed into a neural network. The final training and testing data is a set of arrays of tfidf values for every word in the dictionary based on each review document. Each array of tfidf values represents one review.

Along with the tfidf representations of each review, I also extract the correct semantic label for each review from its file name. Each training and testing document has a label between 1 and 10; anything above 5 is positive and anything below 5 is negative. I convert these values into tuples of positive and negative values. A positive document has a positive element that is closer to +1 and a negative element close to -1, while a negative document has a positive element closer to -1 and a negative element closer to +1. This allows me to train a network that has two outputs: a % certainty that a review is positive and a % certainty that a review is negative.

Now that I have a set of data and label pairs, I can do supervised learning using a neural network. At first I wanted to implement this using PyTorch in Python because many parts are done for me and I just had to put them together. But this would mean that I would have to rewrite the FeatureVector and tfidf parts in python to be able to do the testing portion on new documents. So I opted to write my own implementation of a single-layer neural network using logistic regression in Java.

The neural network is a single fully-connected neuron layer. The number of inputs equals the number of phrases in the dictionary and only two outputs: one for positive certainty and one for negative certainty. The layer is represented by a 2D array that holds weights for every connection between the inputs and outputs. A prediction calculation is done by looping through every output and then summing the product of the inputs and weights between each input and output pair. Then each output is passed through a sigmoid function to clamp the output values between 0 and 1.

To train the network, I implemented a form of logistic regression. First, I shuffle the training data set so that the network would learn positive and negative features evenly. Then I take each set of

training data and calculate a prediction for that data. Then for every output label (positive and negative certainty), I calculate the error of the prediction compared to the expected output. I also calculate the sigmoid derivative of the predicted output. Then for every input to output weight, I multiply the input value, error, sigmoid derivative, and learning rate together to get a value representing how much to change each weight. At the end of each training loop, every weight will be a little better at predicting the correct output.

Finally, to test the network, I pass in a set of training data and label pairs and then do a prediction calculation on each one. As I get each predicted output, I keep track of the total the number of correct guesses. At the end I divide the total by the number of testing documents to get a prediction accuracy.

Evaluation:

The whole process of importing the dictionary, loading each document, calculating tfidfs, extracting labels, training, and testing the neural network takes between 5 to 30 minutes on my PC depending on the number of training iterations and the chosen learning rate.

In the end, the highest accuracy that I could get was 83.3% on the testing data set. This was done using a learning rate of 0.2 after 18 training iterations and all network weights initialized to 0.0001. After 83.3% accuracy is reached, the network bounces around that accuracy but never goes above.

An implementation that uses multiple neuron layers might be able to give more accurate predictions because it could recognize more complex patterns. The single neuron layer is only able to predict based on the number of occurrences of each phrase in a review but a multi-layer network could also recognize relationships between different phrases.

Source Code:

SemanticAnalysis.java:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.ArrayList;

public class SemanticAnalysis {

  public static final boolean LOAD_TFIDF = false;
  public static final boolean LOAD_WEIGHTS = false;
  public static final boolean TRAINING = true;

  public static FeatureVector extractWordFeatures(String reviewPath, ArrayList<String> dictionary) {

    FeatureVector features = new FeatureVector();

    try {
      File reviewFile = new File(reviewPath);

      // Read the review
```

```
        BufferedReader reviewReader = new BufferedReader(new FileReader(reviewFile));
        String reviewLine = reviewReader.readLine();
        while (reviewLine != null) {
          String[] tokens = cleanString(reviewLine).split("[ \t\n]");

          for (int i=0; i<tokens.length; i++) {
            // Make sure the token isn't empty
            if (tokens[i].length() > 0) {
              // Check if three word, two word, or one word phrase exists in the dictionary
              if (dictionary.contains(tokens[i])) {
                if (i < tokens.length-1 && dictionary.contains(tokens[i]+" "+tokens[i+1])) {
                  if (i < tokens.length-2 && dictionary.contains(tokens[i]+" "+tokens[i+1]+"
"+tokens[i+2])) {
                    features.add(tokens[i]+" "+tokens[i+1]+" "+tokens[i+2]);
                    i+=2;
                  } else {
                    features.add(tokens[i]+" "+tokens[i+1]);
                    i++;
                  }
                } else {
                  features.add(tokens[i]);
                }
              }
            }
          }
          reviewLine = reviewReader.readLine();
        }

        reviewReader.close();

      } catch (Exception ex) {
        ex.printStackTrace();
        System.exit(-1);
      }

      return features;
    }

    public static ArrayList<String> importDictionary(String dictPath) {
      ArrayList<String> dictionary = new ArrayList<>();
      try {
        File dictFile = new File(dictPath);

        // Read the whole dictionary file
        BufferedReader dictReader = new BufferedReader(new FileReader(dictFile));
        String dictLine = dictReader.readLine();
        while(dictLine != null) {

          // Split each line into tokens
          String tokens[] = dictLine.split("[ \t\n]");
          String word = tokens[0];
          int value = 0;
```

```
            // Add all words in the dictionary phrase
            for(int i=1; i<tokens.length; i++) {
               // Make sure that the token isn't a semantic value
               try {value = Integer.parseInt(tokens[i]); }
               catch (NumberFormatException ex) {
                  word += " "+tokens[i];
               }
            }

            if (!dictionary.contains(word))
               dictionary.add(word);
            if (!dictionary.contains("not "+word))
               dictionary.add("not "+word);

            dictLine = dictReader.readLine();
         }

         dictReader.close();
      } catch (Exception ex) {
         ex.printStackTrace();
         System.exit(-1);
      }

      return dictionary;
   }

   public static String cleanString(String str) {
      String result = str.toLowerCase();
      result = result.replaceAll("[^a-z\\d-\\s]+", "");
      result = result.replaceAll("  ", " ");
      return result;
   }

   public static double[][] addDocuments(String directoryPositive, String directoryNegative,
ArrayList<FeatureVector> documents, ArrayList<String> dictionary) throws Exception {

      // Get a list of positive and negative files
      String filesPos[] = (new File(directoryPositive)).list();
      String filesNeg[] = (new File(directoryNegative)).list();

      // Generate a list of labels for each document
      double[][] labels = new double[filesPos.length+filesNeg.length][2];
      int labelIndex = 0;

      // Go through all document files in the positive directory
      for (String fName : filesPos) {
         String path = directoryPositive+fName;

         // Get word features of the document
         FeatureVector features = SemanticAnalysis.extractWordFeatures(path, dictionary);
```

```
            // Add features to list of documents
            documents.add(features);

            // Set the label based on filename
            double rating = new Double(fName.split("[_.]")[1]);
            labels[labelIndex][0] = (rating-5.0)/5.0;
            labels[labelIndex][1] = -(rating-5.0)/5.0;

            labelIndex++;
        }

        // Go through all document files in the negative directory
        for (String fName : filesNeg) {
            String path = directoryNegative+fName;

            // Get word features of the document
            FeatureVector features = SemanticAnalysis.extractWordFeatures(path, dictionary);

            // Add features to list of documents
            documents.add(features);

            // Set the label based on filename
            double rating = new Double(fName.split("[_.]")[1]);
            labels[labelIndex][0] = -(5.0-rating)/5.0;
            labels[labelIndex][1] = (5.0-rating)/5.0;

            labelIndex++;
        }

        return labels;
    }

    public static void saveLabels(String fileName, double[][] labels) throws Exception {
        // Open labels file for writing
        File fileLabels = new File(fileName);
        fileLabels.createNewFile();
        FileWriter outLabels = new FileWriter(fileLabels);

        for (int i=0; i<labels.length; i++) {
            // Save the labels to file
            outLabels.write(labels[i][0]+"\n");
            outLabels.write(labels[i][1]+"\n");
        }

        outLabels.close();
    }

    public static double[][] calculateTrainingTfidf(ArrayList<FeatureVector> trainingDocuments,
ArrayList<String> dictionary) throws Exception {

        // Open idf and tfidf files for writing
        File fileIdf = new File("idf_train.txt");
```

```
        fileIdf.createNewFile();
        FileWriter outIdf = new FileWriter(fileIdf);

        File fileTfIdf = new File("tfidf_train.txt");
        fileTfIdf.createNewFile();
        FileWriter outTfIdf = new FileWriter(fileTfIdf);

        double[][] result = new double[trainingDocuments.size()][dictionary.size()];

        // Go through every word in dictionary
        for (int i=0; i<dictionary.size(); i++) {

            // Calculate IDF
            double idf = Tfidf.getIdf(dictionary.get(i), trainingDocuments);
            outIdf.write(idf+" ");

            // Go through every document
            for (int j=0; j<trainingDocuments.size(); j++) {

                // Calculate TF and TFIDF
                double tf = Tfidf.getTf(dictionary.get(i), trainingDocuments.get(j));
                double tfidf = tf*idf;

                // Add tfidf entry to list
                result[j][i] = tfidf;
                outTfIdf.write(tfidf+" ");
            }

            // Write endline for next word
            outIdf.write("\n");
            outTfIdf.write("\n");
        }

        outIdf.close();
        outTfIdf.close();

        return result;
    }

    public static double[][] calculateTestingTfidf(ArrayList<FeatureVector> testingDocuments,
ArrayList<String> dictionary) throws Exception{

        // Open testing tfidf file for writing
        File fileTfIdf = new File("tfidf_test.txt");
        fileTfIdf.createNewFile();
        FileWriter outTfIdf = new FileWriter(fileTfIdf);

        // Open training idf file for reading
        File fileIdf = new File("idf_train.txt");
        BufferedReader idfReader = new BufferedReader(new FileReader(fileIdf));

        double[][] result = new double[testingDocuments.size()][dictionary.size()];
```

```
        // Go through every word in dictionary
        for (int i=0; i<dictionary.size(); i++) {

            // Read training idf from file
            double idf = Double.parseDouble(idfReader.readLine());

            // Go through every testing document
            for (int j=0; j<testingDocuments.size(); j++) {

                // Calculate TF and TFIDF
                double tf = Tfidf.getTf(dictionary.get(i), testingDocuments.get(j));
                double tfidf = tf*idf;

                // Add tfidf entry to list
                result[j][i] = tfidf;
                outTfIdf.write(tfidf+" ");
            }

            // Write endline for next word
            outTfIdf.write("\n");
        }

        outTfIdf.close();
        idfReader.close();

        return result;
    }

    public static double[][] loadTfidf(String fileName, int docNum, ArrayList<String> dictionary) throws
Exception {

        // Open idf and tfidf files for reading
        File fileTfIdf = new File(fileName);
        BufferedReader tfidfReader = new BufferedReader(new FileReader(fileTfIdf));
        double[][] result = new double[docNum][dictionary.size()];

        // Go through every word in dictionary
        for (int i=0; i<dictionary.size(); i++) {

            // Read a line of tfidfs
            String[] tfidfs = tfidfReader.readLine().split("[ \t\n]");

            // Go through every document
            for (int j=0; j<docNum; j++) {

                // Add tfidf entry to list
                result[j][i] = Double.parseDouble(tfidfs[j]);
            }
        }

        tfidfReader.close();
```

```
    return result;
}

public static double[][] loadLabels(String fileName) throws Exception {
    // Open labels file for reading
    File fileLabels = new File(fileName);
    BufferedReader labelsReader = new BufferedReader(new FileReader(fileLabels));
    ArrayList<Double> resList = new ArrayList<>();

    // Read the whole labels file
    String line = labelsReader.readLine();
    while(line != null) {
        resList.add(Double.parseDouble(line));
        line = labelsReader.readLine();
        resList.add(Double.parseDouble(line));
        line = labelsReader.readLine();
    }

    // Convert the [docNum x 2][1] array to a [docNum][2] array
    double[][] result = new double[resList.size()/2][2];
    for (int i=0;i<resList.size(); i+=2) {
        result[i/2][0] = resList.get(i);
        result[i/2][1] = resList.get(i+1);
    }

    return result;
}

public static void main(String[] args) throws Exception {
    // Read the dictionary of terms
    System.out.println("Importing dictionary...");
    ArrayList<String> dictionary = SemanticAnalysis.importDictionary("AFINN-en-165.txt");

    // Create a single layer neural network
    NeuronLayer network = new NeuronLayer(dictionary.size(),2);

    double[][] trainingLabels;
    double[][] testingLabels;
    double[][] trainingData;
    double[][] testingData;

    if (LOAD_WEIGHTS) {
        // Load the old trained weights
        System.out.println("Loading previously trained weights...");
        network.loadNetwork("weights.txt");
    }

    if (TRAINING) {
        if (LOAD_TFIDF) {
            // Load labels
            System.out.println("Loading training labels...");
            trainingLabels = loadLabels("training_labels.txt");
```

```
            System.out.println("Loading testing labels...");
            testingLabels = loadLabels("testing_labels.txt");

            // Load pre-calculated tfidf for every word on every document
            System.out.println("Loading training tfidf for every word...");
            trainingData = loadTfidf("tfidf_train.txt", trainingLabels.length, dictionary);
            System.out.println("Loading testing tfidf for every word...");
            testingData = loadTfidf("tfidf_test.txt", testingLabels.length, dictionary);
        } else {

            // Create an array of feature vectors for every document
            ArrayList<FeatureVector> trainingDocuments = new ArrayList<>();
            ArrayList<FeatureVector> testingDocuments = new ArrayList<>();

            // Add training and testing documents to the corpus and extract their labels
            System.out.println("Adding training documents to corpus...");
            trainingLabels = addDocuments("documents\\training\\pos\\","documents\\training\\neg\\",
trainingDocuments, dictionary);
            saveLabels("training_labels.txt", trainingLabels);
            System.out.println("Adding testing documents to corpus...");
            testingLabels = addDocuments("documents\\testing\\pos\\","documents\\testing\\neg\\",
testingDocuments, dictionary);
            saveLabels("testing_labels.txt", testingLabels);

            // Calculate tfidf for every word on every document
            System.out.println("Calculating training tfidf for every word...");
            trainingData = calculateTrainingTfidf(trainingDocuments, dictionary);
            System.out.println("Calculating testing tfidf for every word...");
            testingData = calculateTestingTfidf(testingDocuments, dictionary);
        }

        // Test the starting weights
        double topAccuracy = network.getAccuracy(testingData, testingLabels);
        System.out.println("Starting test accuracy: " + topAccuracy + "%");

        // Iterate 100 times
        System.out.println("Training...\n");
        for (int i = 0; i < 25; i++) {
            // Train the network
            network.train(0.2, trainingData, trainingLabels);

            // Test the prediction accuracy on testing data
            double testAccuracy = network.getAccuracy(testingData, testingLabels);
            System.out.println("Test accuracy " + i + ": " + testAccuracy + "%");

            // Save the network if it's the best accuracy so far
            if (testAccuracy > topAccuracy) {
                network.saveNetwork("weights.txt");
                topAccuracy = testAccuracy;
            }
        }
```

```
            System.out.println("Top test accuracy: " + topAccuracy + "%");

        } else {
            // Load single review to test
            ArrayList<FeatureVector> finalDocuments = new ArrayList<>();
            finalDocuments.add(extractWordFeatures("review.txt", dictionary));
            double[][] finalData = calculateTestingTfidf(finalDocuments, dictionary);

            // Predict the semantic value of the review
            double[] outputs = network.calculate(finalData[0]);
            System.out.println("Prediction: "+String.format("%.2f",outputs[0]*100.0)+"% Positive and
"+String.format("%.2f",outputs[1]*100.0)+"% Negative");
        }
    }
}
```

FeatureVector.java:

```
import java.util.ArrayList;

public class FeatureVector {

    private Feature[] alphabet = new Feature[26];
    private int numFeatures = 0;

    public void add(String word) {
        add(word, 1);
    }

    public void add(String word, int value) {
        int pos = word.toLowerCase().charAt(0)-'a';

        // Look an existing feature with that name
        boolean found = false;
        Feature node = alphabet[pos];
        while(node != null && !found) {
            if (node.word.equals(word)) {
                node.count++;
                found = true;
            } else {
                node = node.next;
            }
        }

        if (!found) {
            // Check if the list is empty
            if (alphabet[pos] == null) {
                alphabet[pos] = new Feature(word, value);
            } else {
                // Link a new feature to the front of the list
                Feature addition = new Feature(word, value);
```

```
            addition.next = alphabet[pos];
            alphabet[pos].prev = addition;
            alphabet[pos] = addition;
        }
    }

    // Increase features counter
    numFeatures++;
}

public void remove(String word) {
    int pos = word.toLowerCase().charAt(0)-'a';

    // Look for the word
    Feature node = alphabet[pos];
    while(node != null && !node.word.equals(word)) {
        node = node.next;
    }

    // Remove the word from the list of features
    if (node != null) {
        numFeatures -= node.count;

        if (node.prev != null)
            node.prev.next = node.next;
        if (node.next != null)
            node.next.prev = node.prev;
        if (node == alphabet[pos])
            alphabet[pos] = node.next;
    }
}

public Feature get(String word) {
    int pos = word.toLowerCase().charAt(0)-'a';
    Feature node = null;

    // Only search for words starting with an alphabetical character
    if (pos >= 0 && pos < 26) {
        // Look for the word
        node = alphabet[pos];
        while (node != null && !node.word.equals(word)) {
            node = node.next;
        }
    }

    return node;
}

public int size() {
    return numFeatures;
}
```

```
    public ArrayList<Feature> toArrayList() {
        ArrayList<Feature> list = new ArrayList<>();

        // Traverse the array or hashed lists
        for (int i=0; i<alphabet.length; i++) {

            // Go through every feature in the linked list
            Feature node = alphabet[i];
            while(node != null) {

                // Add the feature to the result list
                list.add(node);
                node = node.next;
            }
        }

        return list;
    }

    public void print() {
        System.out.println("Feature Vector:");

        for (int i=0; i<alphabet.length; i++) {

            System.out.print((char)(i+'a')+"{");

            Feature node = alphabet[i];
            while(node != null) {
                System.out.print("("+node.word+":"+node.count+"),");
                node = node.next;
            }

            System.out.print("}\n");

        }
    }
}
```

Feature.java:

```
public class Feature {
    public Feature next;
    public Feature prev;

    public String word;
    public int count;

    public Feature(String word, int value) {
        this.word = word;
        this.count = value;
```

```
    }
}
```

Tfidf.java:

```
import java.util.ArrayList;

public class Tfidf {

    public static double getTf(String term, FeatureVector document) {
        // Get count of term in document
        Feature word = document.get(term);
        double count = (word == null) ? 0:word.count;

        // Get total features in document
        double total = document.size();

        // Return 0 if the document is empty
        if (total == 0)
            return 0.0;

        // Calculate tf
        return count/total;
    }

    public static double getIdf(String term, ArrayList<FeatureVector> documents) {
        // Get total number of documents
        double docNum = documents.size();

        // Go through each document to find term
        int count = 1;
        for (FeatureVector feats : documents)
            if (feats.get(term) != null)
                count++;

        // Calculate IDF
        return Math.log(docNum/count);
    }

    public static double getTfIdf(String term, FeatureVector document, ArrayList<FeatureVector>
documents) {
        double tf = getTf(term, document);
        double idf = getIdf(term, documents);

        // Calculate TFIDF
        return tf*idf;
    }
}
```

NeuronLayer.java:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class NeuronLayer {

   public double[][] weights;
   public int numInputs;
   public int numOutputs;

   public NeuronLayer(int numInputs, int numOutputs) {

      this.numInputs = numInputs;
      this.numOutputs = numOutputs;

      weights = new double[numInputs][numOutputs];

      // Go through every input neuron
      for (int inputIndex = 0;inputIndex < numInputs;inputIndex++) {
         // Go through every output neuron connected to the current input neuron
         for (int outputIndex = 0;outputIndex < numOutputs;outputIndex++) {
            // Start the weight at a number close to 0
            weights[inputIndex][outputIndex] = 0.0001;
         }
      }
   }

   public double[] calculate(double[] inputs) {
      // A place to put the output values
      double[] outputs = new double[numOutputs];

      // Go through every output neuron
      for (int outputIndex = 0; outputIndex < numOutputs; outputIndex++) {
         // Go through every input neuron connected to the current output neuron
         for (int inputIndex = 0; inputIndex < numInputs; inputIndex++) {
            // Calculate the effectiveness of each input neuron on the current output neuron
            outputs[outputIndex] += weights[inputIndex][outputIndex] * inputs[inputIndex];
         }

         // Apply the sigmoid curve to every output to clamp it between 0 and 1
         outputs[outputIndex] = 1 / (1 + Math.exp(-outputs[outputIndex]));
      }

      return outputs;
```

```
  }

  public void train(double learnRate, double[][] inputData, double[][] outputData) {

    // Shuffle the input data
    Integer[] setRange = new Integer[inputData.length];
    List<Integer> setList = Arrays.stream(IntStream.rangeClosed(0, inputData.length-
1).toArray()).boxed().collect(Collectors.toList());
    Collections.shuffle(setList);
    setList.toArray(setRange);

    // Go through every training set
    for(int set = 0; set < inputData.length; set++) {

      // See what the network thinks of the input set
      double[] outputs = calculate(inputData[setRange[set]]);

      //Create an empty set of adjustments
      double[][] change = new double[numInputs][numOutputs];

      //Go through every output neuron
      for(int outputIndex = 0; outputIndex<numOutputs; outputIndex++) {

        //Calculate the error and sigmoid derivative of every output
        double error = outputData[setRange[set]][outputIndex] - outputs[outputIndex];
        double sigmoidDerivative = outputs[outputIndex] * (1 - outputs[outputIndex]);

        //Go through every input to calculate how much to change it
        for(int inputIndex = 0; inputIndex < numInputs; inputIndex++) {

          //Calculate the amount to change using change = input * error * sigmoid_derivative(output)
          weights[inputIndex][outputIndex] += inputData[setRange[set]][inputIndex] * error *
sigmoidDerivative * learnRate;
        }
      }
    }

  }

  public boolean test(double[] inputData, double[] outputData) {
    // Get a prediction for a set of data
    double[] outputs = calculate(inputData);

    // Figure out which prediction label has the highest % certainty
    double biggest = 0;
    int answer = -1;
    for (int i=0; i<outputs.length; i++) {
      if (outputs[i] > biggest) {
        biggest = outputs[i];
        answer = i;
      }
    }
```

```
    // Figure out which expected label has the highest % certainty
    biggest = 0;
    int expected = -1;
    for (int i=0; i<outputs.length; i++) {
      if (outputData[i] > biggest) {
        biggest = outputData[i];
        expected = i;
      }
    }

    // Return whether the predicted and expected labels are the same
    return (answer == expected && answer != -1 && expected != -1);
  }

  public double getAccuracy(double[][] testingData, double[][] testingLabels) {
    // Count the number of correct predictions
    double numRight = 0;
    for (int i=0; i<testingData.length; i++)
      numRight += test(testingData[i], testingLabels[i]) ? 1:0;

    // Return % accuracy of the network
    return (numRight/testingData.length)*100;
  }

  public void saveNetwork(String fileName) throws Exception {
    // Open the weights file for writing
    File fileWeights = new File(fileName);
    fileWeights.createNewFile();
    FileWriter outWeights = new FileWriter(fileWeights);

    // Save weight value for every input-output pair
    for (int i=0; i<numInputs; i++) {
      for (int j=0; j<numOutputs; j++) {
        outWeights.write(weights[i][j]+"\n");
      }
    }

    outWeights.close();
  }

  public void loadNetwork(String fileName) throws Exception{
    // Make sure the weights file exists
    File reviewFile = new File(fileName);
    if (!reviewFile.exists())
      return;

    // Read the weights file
    BufferedReader reviewReader = new BufferedReader(new FileReader(reviewFile));
    String reviewLine = reviewReader.readLine();
    int word = 0;
```

```
        // Read until the end of the file
        while (reviewLine != null) {
            // Go through each label
            for (int j=0; j<numOutputs; j++) {
                // Load the weight for the word and label
                weights[word][j] = new Double(reviewLine);
                reviewLine = reviewReader.readLine();
            }
            word++;
        }

        reviewReader.close();
    }

    public void print() {
        // Go through every input neuron
        for (int inputIndex = 0; inputIndex < numInputs; inputIndex++) {
            // Go through every output neuron connected to the current input neuron
            for (int outputIndex = 0; outputIndex < numOutputs; outputIndex++) {
                // Print the current weight between the two neurons
                System.out.println("Weight "+inputIndex+" to "+outputIndex+":
"+weights[inputIndex][outputIndex]);
            }
        }
    }
}
```