Pride and wallet still smarting from his meeting with Alyssa, Ben returns home and plots his revenge. “I’ll fix her,” Ben mutters, pacing about his bedroom. “She’s not so great. She even stuck Louis and Eva at a private table at her dinner party, and while I didn’t tell her I wanted to get to know Eva better, she should have figured it out.”

After pondering the matter in his slow way for quite some time, Ben decides that the right way to teach Alyssa a lesson is to design software that can analyze political situations better than Alyssa herself can. Remembering his 6.034 days, Ben decides to start by writing some functions to compute distance in various ways, so he can use them in his political classifiers.

**Problem 1: Distance Metrics**

For this problem, you are to implement the Euclidean distance metric and the generalized Hamming distance for vector inputs. You will use these later on in the problem set. For our purposes, the generalized Hamming distance is the sum of the dissimilarity terms of two inputs, where the dissimilarity term from two corresponding components of the inputs is 1 if they have different values and 0 otherwise. You can assume the elements of the lists for the hamming function are not list structured. See the public test cases for disambiguating examples.

Warmed up and with distance metrics ready, Ben digs up some nearest neighbors code he wrote way back in the day.

“Ah, I remember hacking this together. The 1980s. Those were strange times. Bad haircuts, worse music. The gin was good, though. I hope that doesn’t mean the code is full of bugs.”

Mentally still stuck in the 1980s, Ben decides to test out his political prediction programs on the database of congressional voting records he collected during those days, when he thought he was going to be World President (he gave up on this dream mostly because World President doesn’t exist).

**Problem 2: Congressional Prediction: NN with Hamming**

For this problem, you are to use nearest neighbors on the congressional voting records dataset to predict the party membership of particular congressmen. Use your hamming distance function as the distance metric for nearest neighbors. Train using congress-training-data.txt and test using congress-test-data.txt. Look at congress-info.txt for an explanation of the dataset, and nn-example.scm for example code that uses the nearest neighbors framework.

NOTE: If the official, supported scheme (from the scheme locker) throws out-of-memory errors, start scheme by running

```
scheme -stack 6000 -heap 4000.
```

We cannot help you if you are using an unofficial configuration.

1. What party does nearest-neighbors predict for the first congressman in the test set?
2. What is the accuracy of the nearest-neighbors classifier on the test set (the fraction, expressed as a decimal number, of the number correctly classified from the number of total elements)?

Ben was quite surprised by the accuracy of this simple strategy. He wonders for a moment if it was simply a matter that the 80s were simpler times. He then decides that he might be able to do even better if he used a different distance metric - one which didn’t count “maybe” as a particular vote. He decides to represent examples in a Euclidean space and avoid all the weirdness of Hamming distances altogether.

**Problem 3: Son of a Congressional Prediction: NN with Euclid**

*Answer the same questions as you did for the previous problem, except with a nearest neighbors classifier using the Euclidean distance function you defined previously. Base your code to call the classifier on the code provided in nn-example.scm. Use the procedure congress-to-metric (from congress-utils.scm) to convert the provided datasets to lists of numbers 1/-1/0 before running your classifier. (If you are confused, run the function on e.g. test-data or train-data from nn-example.scm and look at the results, then think about what input your Euclidean distance function expects.) You may find the results surprising; think about why they could come about.*

Surprised by the results of this last attempt, Ben meditates on the relationship between correlation and causation. He briefly considers taking a hiking trip to the Whites so he can climb a peak and ponder the issue properly, but remembering his goal - embarassing Alyssa - he decides to think about some issues involving nearest neighbors first.

**Problem 4: Nearest Neighbor Concepts**

*For this problem, provide a 'yes or 'no answer to each question.*

1. Did the use of a Euclidean distance metric improve the performance of a nearest-neighbors-based classifier for congressional voting records?

2. Would you expect nearest neighbors to perform well in a situation where the true class of a given point is chosen randomly according to a uniform distribution over classes?

3. Would you expect nearest neighbors to perform well (say, over 80% accuracy on the test set) on the congress dataset when it is only given the first record to train on?

Fed up with nearest neighbors in the political arena, Ben digs up his 6.034 notes and hits upon identification trees. "Surely these can be used to classify politicians!" Ben thinks. "ID trees also contain clear explanations of why they make decisions, so hopefully there will be fewer surprises in their results - and maybe I’ll even learn something about the domain of politics."

**Problem 5: The Return of Congressional Prediction: ID Trees**

*For this problem, use the provided ID tree code (in id-congress.scm) to train ID trees based on the congressional voting record dataset. Examine the tree that results by looking at the value of the symbol congress-idtree (and paying attention to the comments).*
NOTE: You will be expected to know how to simulate the ID tree algorithm on small datasets (typically, though not exclusively, involving continuous attributes) for quiz 2 and the final exam. The portion of the problem set that drilled you on this was removed to lessen the time burden. You may want to take some time out here to practice constructing ID trees based on small examples, and in particular acquaint yourself with the average disorder formula, so that you are not caught off guard.

1. What issue initially splits the examples into groups with least average disorder? (Respond with its full string name.)

2. Did an ID tree trained on the first 100 examples perform better than nearest neighbors?

3. Examine the decision tree produced by the ID tree algorithm suite on this data. Does it make sense to you? (Nothing to submit for this part.)

"Identification trees are all well and good," thought Ben to himself, "but they seem... limited. What if I tried something with much more general structure. Like neural networks! My brain is made of neurons, right?" After looking up "brain" in the encyclopedia, Ben continues his train of thought. "Maybe these artificial neural networks could do something for me here. If only I had a working implementation."

Problem 6: Implementing Neural Networks

For this problem, you are to complete the provided implementation of sigmoidal neural networks trained via backpropagation. You must fill in the forward-prop and backward-prop procedures. forward-prop returns a list of activations at each layer in the network (including a copy of the inputs, in the first element). backward-prop returns a list of the errors at each layer, including derivative terms (e.g. \((f' \ast (\text{true-out} - \text{computed-out}))\) at the last layer). If you like, use the procedures in bpnn-helper.scm to simplify things throughout this problem, by expressing the computations in matrix notation. Pay attention to the comments in ps4.scm.

NOTES ABOUT THIS PROBLEM: Realise that according to the specification, backward propagating neural nets (bpnn's) can have multiple (hidden) layers and multiple outputs. You are to implement the forward and backward propagation functions for this general case. Also note that as far as the public test cases go, this problem will be tested by problems 8 and 9; you should make sure to test your solutions by hand. These routines will receive substantial weight in the hidden test cases (to discourage wild guesses for 8 and 9).

To emphasize the spec, forward-prop is supposed to learn a list of network outputs at each layer - that is, a list of vectors, where each vector corresponds to a layer, and the ith element in that vector corresponds to the activation (output) of the ith node in that layer. Also note that backprop returns the list of error vectors; if \(\delta_j\) is the error vector for layer j, the correspondence to Winston’s formulas is: \(\delta_j = B_k f'(o_k)\) where \(o_k\) is the activation vector for layer k (output vector in Winston’s notation) and \(f'\) is the derivative of the sigmoidal function. This representation of the weight update formula is more standard. If you are confused, we suggest you go through one of the justifications for Winston’s formula; the correspondence with this one will be clear, and this will be a good exercise.

Ready to start testing his neural network implementation, Ben decides to test it out on a simple (yet classic) function: XOR.

Problem 7: Neural Networks XOR Example
For this problem, you are to apply your BPNN framework to learning the XOR function, and report on the results. Use the procedures in bpnn-helper.scm to help you throughout this problem. Once your forward and backward propagation code is done, all the remaining code is just simple bookkeeping procedures (repeatedly calling the training functions, etc).

1 Generate 10 networks according to the following conditions, and report the average of the average test errors for each network (e.g. if the average test error for run 1 was 0.05, the average test error for run 2 was 0.06, and so on, you’d report (0.05 + 0.06 + ...) / 10) and the average number of epochs needed. You will want to write a general procedure that instantiates a bpnn with a specified init function, trains it with a specified trainer (and training parameters), and stops it for a specified set of conditions. You will want another general procedure that calls this one for a set number of tries and computes the average number of epochs and the average of the average of the test errors. You’ll be running several experiments on bpnn’s and really don’t want to have to do the tests by hand.

– Your network should have two input nodes, two hidden nodes, and one output node.
– Generate initial weights for your network randomly from a uniform distribution over the interval [-1,1].
– Train your network using the train-online-random function using the xor-data dataset. (This training function randomly chooses which example to use at each step.).
– Use a learning rate (α) of 0.1. Train for 1000 steps per training epoch.
– After each epoch, check to see if training should stop. Stop training if 10 or more epochs have passed, if the maximum error on the xor function is 0.1, or if the average squared error on the training set has decreased by less than 0.01.

2 Repeat the experiment with a learning rate of 0.4.

3 Repeat the experiment with a learning rate of 0.1, constant initial weight values (of 1.0) and with training instead done by the train-online-deterministic function. (This training function cycles through the examples one-by-one.

4 Meditate on the art of training neural networks via backpropagation. (Nothing to turn in.)

Pleased by his eventual success at training an XOR function, Ben decides to take a break. Musing on the complexities he discovered surrounding backpropagation parameters, he decides to give Eva a call to see if she can offer any advice. She doesn’t pick up, and instead Ben hears the following message:

"Hello. You have reached Eva Lu Ator. If you are not Ben Bitdiddle, please leave a message after the beep. If you are Ben Bitdiddle, don’t bother - my answering machine is programmed to detect your voice and hang up immediately. And no, your Swedish Chef impression will not fool it."

Hanging up the phone immediately, Ben decides there’s nothing left to do but try to finish the congressional prediction task using backpropagation networks. Thinking ahead (and realising overfitting is likely to pose a problem in this case), Ben decides to try something he read about in a machine learning book.

Instead of stopping when the training error reaches a minimum (or doesn’t change, or the search times out), Ben decides he will save a portion of his training data for testing purposes. His idea is to avoid overfitting by training on, say, the first half of the known data, but testing on the second half, and stopping when the error on that second, known half reaches a minimum. He recalls that this technique is called cross-validation and remains one of the most important techniques for tuning parameters in pattern recognition systems (for more information on this, take 9.914 or 6.867). This way, though he loses the information in the validation set, he’ll hopefully stop before overfitting.

**Problem 8: Will Congressional Prediction Ever End?: BPNNs**

4
For this problem, you are to apply your BPNN framework to the congressional party prediction problem. Formalize a yes vote as a 1, and no vote as a -1, and a maybe vote as a 0. Formalize republican party membership as a 1 and democratic membership as -1. (The procedure congress-to-bpnn-metric in congress-utils.scm will do this for you, converting a dataset like train-data or test-data from the nn problem into one suitable for use with bpnns.)

NOTE: You may find the following fragment useful in setting up the problem:

(load "congress-utils")
(define train-data (load-congress-dataset "congress-training-data.txt"))
(define test-data (load-congress-dataset "congress-test-data.txt"))

(define bpnn-train (list-head (congress-to-bpnn-metric train-data) 50))
(define bpnn-test (congress-to-bpnn-metric test-data))

;; These are the training and validation
(define real-train (list-head bpnn-train 25))
(define real-valid (list-tail bpnn-train 25))

• Generate and train randomly (weights uniform on [-1,1]). Learning rate 0.1, epochs 1000 steps, stop when max error is under 0.1, change in average is less than 0.01, or more than 10 epochs pass. Use the first 25 training examples to train the network, and the next 25 (that is, examples 26-50) for validation. Report the average of the average squared errors and the average number of epochs needed for stopping assuming a network with 3 hidden layers.

Problem 10: Wrap-up

Make sure your code passes the public test cases. Use the same testing protocol you’ve used for the previous problem sets.