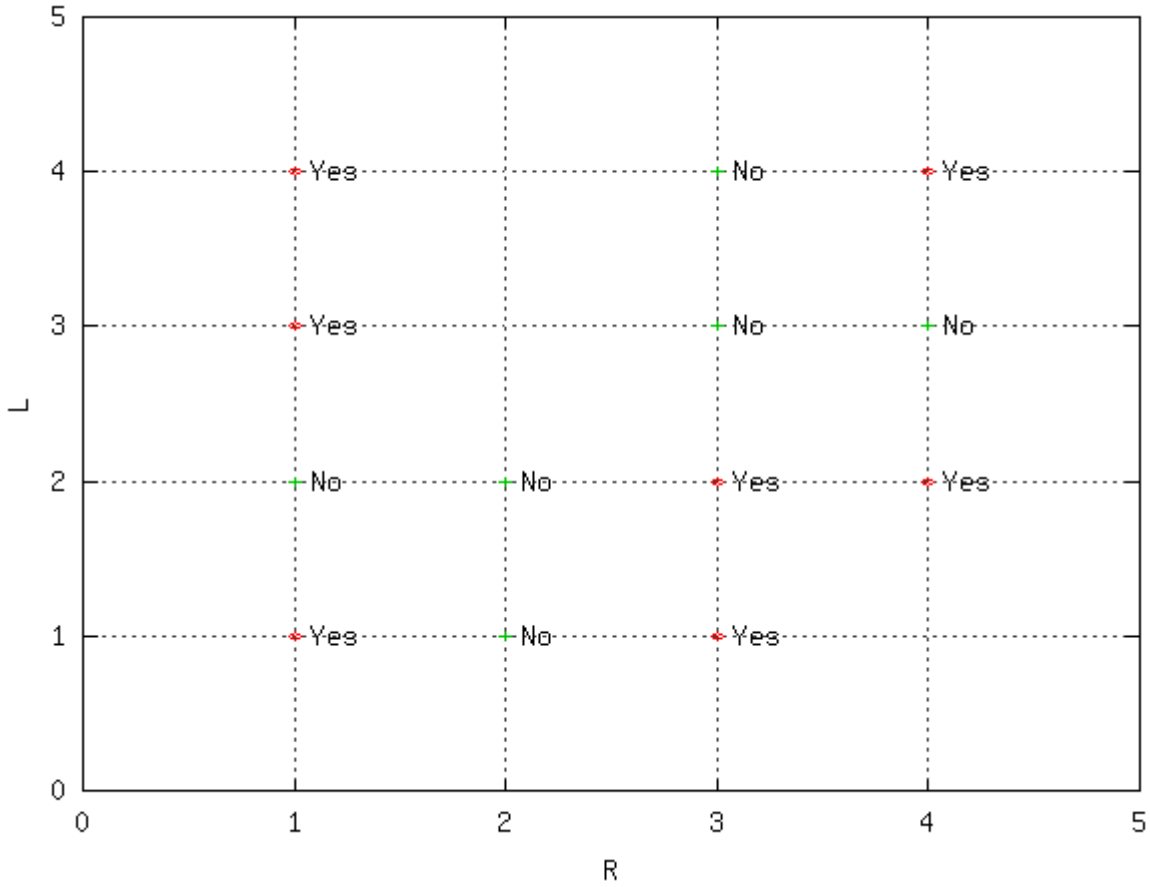
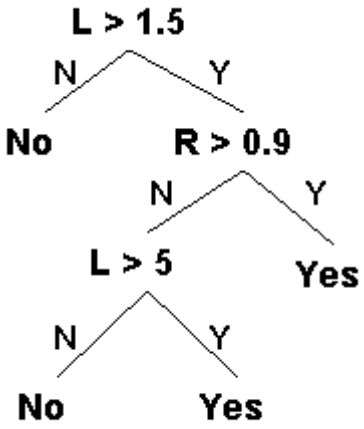
1. Which of the following are symptoms of overfitting?
    a. ☐ The performance on the training set is perfect in a noisy domain.
    b. ☐ You have two x's with the same y in the training set.
    c. ☐ Performance on a test set is about the same as on the training set.
    d. ☐ Performance on a test set is much better than on the training set.
    e. ☐ Performance on a test set is much worse than on the training set.

At the exit of the movies, you did a survey to find whether people who liked the movie "hulk" also liked "x-men united". You got the data set below:

Given the data in the figure below:



construct a decision tree, using the greedy algorithm we have described. Assume the tests are of the form $X > a$, where a is the midpoint between two feature values. The left branch is the ``false" branch and the right is the ``true" branch. Enter your answer below by listing the contents of the nodes in the order that they would be expanded by a depth-first search of the tree. For example, the following tree:



would be entered as:

1. L > 1.5
2. No
3. R > 0.9
4. L > 5
5. No
6. Yes
7. Yes

For each non-leaf node, also enter the average entropy (accurate to two decimal places). For leaf nodes, enter the class ``Yes" or ``No" and an average entropy of 0.0.

1. Node: _____ Average Entropy: _____
2. Node: _____ Average Entropy: _____
3. Node: _____ Average Entropy: _____
4. Node: _____ Average Entropy: _____
5. Node: _____ Average Entropy: _____
6. Node: _____ Average Entropy: _____
7. Node: _____ Average Entropy: _____
8. Node: _____ Average Entropy: _____
9. Node: _____ Average Entropy: _____
10. Node: _____ Average Entropy: _____
11. Node: _____ Average Entropy: _____
12. Node: _____ Average Entropy: _____
13. Node: _____ Average Entropy: _____

14. Node: [ ] Average Entropy: [ ]
15. Node: [ ] Average Entropy: [ ]

Note that this is not the simplest possible tree for this data, but it's what the greedy tree-builder finds.

Nearest neighbors and identification trees are simple memory based solutions. We can learn more complex functions with neural nets. First, we need to understand how the basic unit of a neural net, a perceptron functions. Note that a perceptron differs from a sigmoid unit by the threshold function it uses. In this question, we are going to look at a perceptron with a step threshold function.

We'll look at the behavior of the perceptron algorithm running on the following data set – say something about the data set – they represent movie goers 1-8 who have seen movies x1-x3. This is to classify the people into classes based on their tastes:

i	x_1^i	x_2^i	x_3^i	y^i
1	0	0	0	+1
2	1	0	0	+1
3	0	1	0	+1
4	0	0	1	+1
5	1	1	0	-1
6	0	1	1	-1
7	1	0	1	-1
8	1	1	1	-1

Assume that the initial values of the weights in the perceptron algorithm are **w**=[-0.5 0 -1 0], where the first weight corresponds to the offset of the separator. Assume we are using a rate of 1.

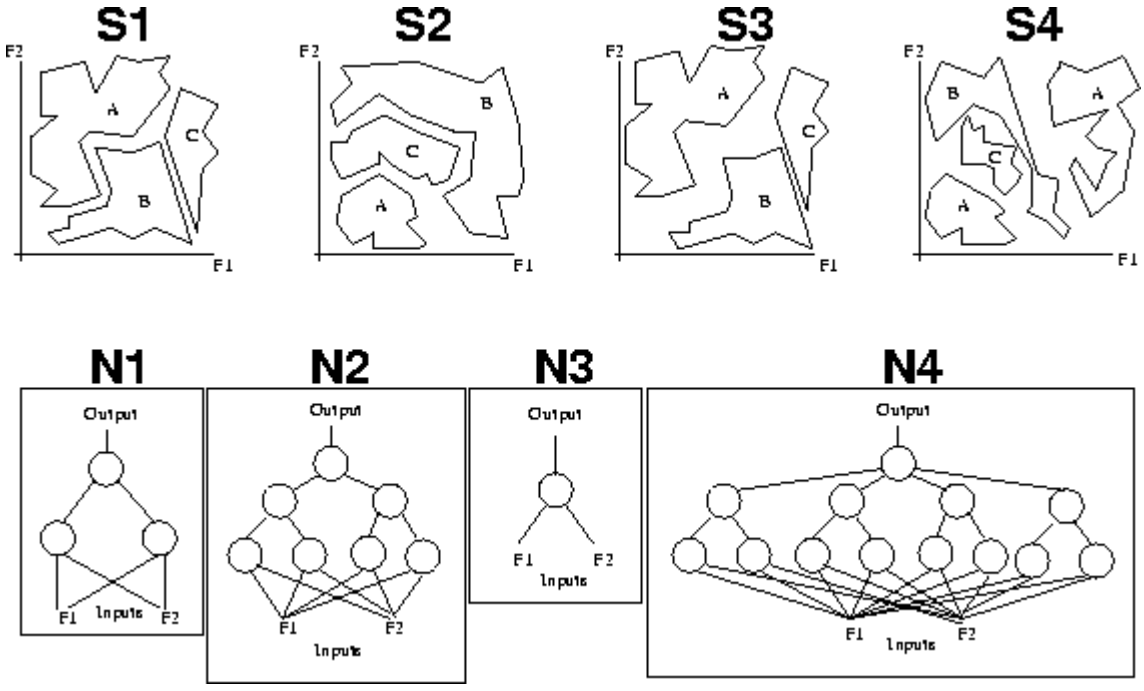
Assume the algorithm goes through the points in the data set in order. For each of the points in the data set, write the weight vector after that point is examined in the inner loop of the algorithm. The weight changes are cumulative just like in the algorithm. If the weight does not change after examining a point, just copy the same weight vector from the previous point.

1. After point 1: [ ]
2. After point 2: [ ]
3. After point 3: [ ]
4. After point 4: [ ]

5. After point 5: [ ]
6. After point 6: [ ]
7. After point 7: [ ]
8. After point 8: [ ]

Recently when going through Prof. Frank N. Stein's (a former 6.034 professor) papers it was found that the TA who did the grading of the quizzes, Mr. Percy Eptron, was in fact a neural network program written several years previously by one of Stein's UROP students. Stein had been experimenting with altering the structure of `Percy' to improve the performance. The Percy network was built from perceptron units that applied a threshold function to the weighted sum of their inputs, and returned either 0.0 or 1.0. The network was supposed to return 1.0 for an A-grade student, and 0.0 for anyone who got grades B or C.

It was clear that Stein had been studying the effect the number of units, and their arrangement into layers, had on the ability to learn. In the following figure, the top row shows four figures of 2-dimensional feature-spaces, S1 to S4. Each feature-space has regions marked A, B, and C. The bottom row shows four perceptron networks, N1 to N4.

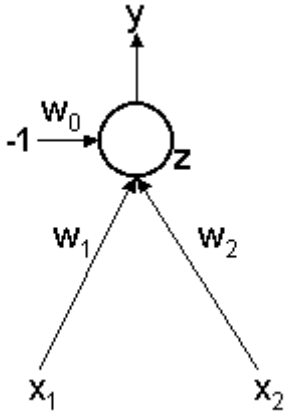


For each feature-space-region figure, enter the label of the **simplest** perceptron network from among the choices given that could distinguish A-grade students from those with

lower (not-A) grades without any misclassifications. (Note: if you want to, you can use the same network more than once).

1. S1: No Answer
2. S2: No Answer
3. S3: No Answer
4. S4: No Answer

Consider the simple neural net sigmoid unit below:

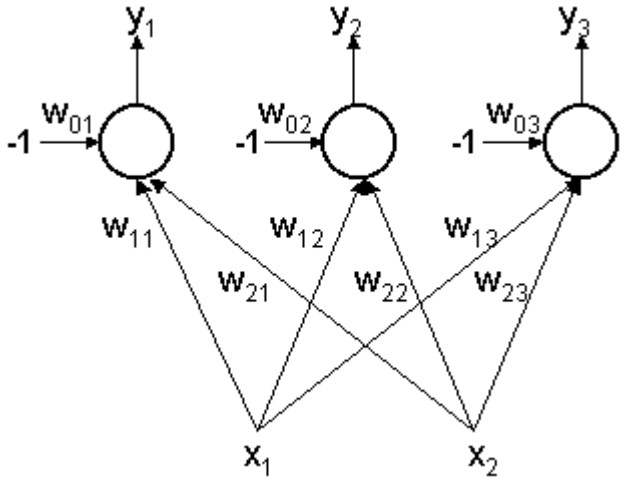


Assume that all the weights start out zero. Assume the input is x_1=0.1 and x_2=1 and the desired output is 1. For all the questions whose answers are numbers, enter real numbers rounded to 2 decimal digits.

1. What is the value of the actual output with the initial conditions described above?
2. Assuming a learning rate of 8 and the standard backpropagation algorithm, what would be the new value of w_0?
3. Assuming a learning rate of 8 and the standard backpropagation algorithm, what would be the new value of w_1?
4. Assuming a learning rate of 8 and the standard backpropagation algorithm, what would be the new value of w_2?
5. What would be the value of the actual output with these updated weights?

Here you'll use a neural network to solve a three-class classification problem of the type illustrated in the figure below. To do this we can introduce three output neurons and predict the label based on which output is the largest. So, for example, if the input (x_1,x_2) generates the outputs y_1, y_2, and y_3 in such a way that y_2 is the largest, then we would

assign the label ``2" to the input example. A simple neural net, with no hidden layer that we'll use for this purpose is given in the figure below.



You can assume that the activation functions (also known as threshold functions, also known as transfer functions) for the three neurons are the typical sigmoid (squashed S) function. The output of neuron i is computed by first evaluating the total input to the neuron, $z_i = w_{1i} x_1 + w_{2i} x_2 - w_{0i}$, and passing this through the sigmoid function to derive the output:
y_i=sigmoid(z_i).
Recall that dsigmoid(z)/dz is y*(1-y).

Assume that all the **weights are initially zero** and that the error function is the **squared error** :

$$E=1/2[(y_1 - y_1^*)^2]+ 1/2[(y_2 - y_2^*)^2]+ 1/2[(y_3 - y_3^*)^2]$$

1. To train this neural net to solve the three-way classification problem, we must establish a correspondence between the labels and specific targets values for the output neurons. In the table below, provide an appropriate translation into desired targets of **0** or **1**.

label	y_1^*	y_2^*	y_3^*
1			
2			
3			

2. The network is then presented with the following example:
   inputs = (x_1,x_2) = (1, -1)
   target label = 3
   Using your translation and by setting the **learning rate equal to 8**, write the

values of the weights after updating using the backpropagation rulen with the single training example.

w_{01} w_{02} w_{03} w_{11} w_{12} w_{13} w_{21} w_{22} w_{23}

3. Now, let's inspect what the network has learned on the basis of the single example so far. In the table below, indicate the labels that the network would predict for the input examples. If a single unambiguous label is predicted, enter the label number. Use, for example, ``1 2 3" (numbers separated by spaces) to denote a tie between all the labels and analogously for other possible ties.

input example (x_1,x_2)	predicted label (1,2, or 3)
(1,-1)	
(1,0)	
(1,1)	
(-1,1)	

For each of the questions below, check all that apply.

1. We select small random weights to start neural net training so as to:
   a. ☐ eliminate the need for a momentum term
   b. ☐ avoid saturation of the sigmoid units
   c. ☐ obtain a linear classifier
2. stopping training before a neural net has achieved minimal error on the training set will typically:
   a. ☐ reduce training error accuracy
   b. ☐ reduce overfitting
   c. ☐ reduce generalization accuracy
3. Given a validation set (a set of samples which is separate from the training set), you could use it to:

a. ☐ choose the number of hidden units in a neural network

b. ☐ decide when to stop training

c. ☐ choose among two sets of weights for a neural net

Genetic Algorithms:
Part 1: Evolving Strings

Let's start by looking at the problem of ``evolving" strings to match some given string. We start out with a population of random strings and apply the mutation operators to produce new strings.

To get a feeling for the operation of the code, for each of the cases below, run ten experiments of the GA code, with an initial population of 10 and allowing up to 1000 steps before it gives up, that is, (run-ga 10 10 1000). Report the number of successful trials, that is, the string is found and the average number of mutations reported at the end of the run.

Note that your results will be probabilistic, so you may want to run the set of experiments several times, especially if your answers do not check out at first.

If you see stack overflow errors running in MIT Scheme, try starting Scheme with options
-stack 500 -heap 3200.

1. Target string = ``target'. Do (init-strings "target") before run-ga
   Number of succesful trials =
   [No response. The valid answer is: 10]
   Average number of mutations =
   [No response. The valid answer is: "Between 230 and 540"]
2. Target string = ``intelligent". Do (init-strings "intelligent") before run-ga
   Number of succesful trials =
   [No response. The valid answer is: 9]
   Average number of mutations =
   [No response. The valid answer is: "Between 670 and 1090"]
3. What is the size of the search space that is being searched in this last case? That is, how many possible strings are there in total? Enter smallest integer x such that 10^x > number-possible-strings.
   [No response. The valid answer is: 16
   The actual answer is 26^{11}]

Part 3: Smart Mutate

The classic mutation operator tends to produce a lot of invalid offspring in the TA/course domain. Your job here is to produce a new mutation operator that is more likely to

produce valid offspring, that are still different from the parent. The code will be graded by how well it succeeds in doing that task. The test gives you points for different benchmark levels of performance on 50 attempted mutations. The standard mutation operator gets around 10.

We encourage you try your mutation code on the TA problem you tried above and see how well it does relative to the standard mutation operator. Replacing the standard mutation operator with the improved one we give as our answer, we always get 10/10 succesful runs with an average of 140 mutations per run. These ``mutations'' are more complex than the original ones, but it is still a remarkable improvement.

```
;; This mutate operator attempts to propagate changes so that the course
;; gene vector stays valid.
;; It performs well on the given test of a population of size 10 and
;; quiescence threshold of 1000 - as stated, it always succeeds, with
;; 140 mutations per run on average.
;;
;; A common solution is to keep mutating until the mutated gene vector
;; is both different and valid. While this shows an understanding of the
;; problem, it is non-optimal. There is a lot of work to be done for
;; that, and it can potential go on for a long time. Perhaps more
;; importantly, it will only allow point mutations - mutations in one
;; gene at a time. That will limit the distance you can move in the
;; genotype space at each generation, and may even make some gene
;; vectors unreachable (such as when two TAs need to be swapped to
;; stay valid). The solution here will allow large moves in 'valid'
;; directions.

;; Tries to get a consistent mutation or just returns the parent
(define (mutate course-assignment)
  (or (smart-mutate course-assignment)
      course-assignment))

;; smart-mutate:
;;   Randomly change one course gene to another valid gene for that
;;   course. Propagate the changes by making new random genes to avoid
;;   any conflicts created. If this fixing-up fails, returns original genome;
;;   otherwise, returns the new course gene vector.
;;  course-genes: the original course gene vector
;;  max-spread-arg:     an optional argument giving the maximum number of
;;                      conflict-resolving propogations to attempt. Defaults
;;                      to the length of the gene vector.
(define (smart-mutate course-genes . max-spread-arg)
  (let ((max-spread (if (null? max-spread-arg)
                        (vector-length course-genes)
                        (car max-spread-arg))))
```

```
   (let* ((zap-course (random (vector-length course-genes)))
          (possible-genes (vector-ref *alphabet* zap-course)))
     (if (= 1 (length possible-genes))       ; no choices for this course
         (smart-mutate course-genes max-spread) ; try again
         (let ((old-gene (vector-ref course-genes zap-course))
               ;; pick a new ta for some course
               (new-gene (list-ref possible-genes
                                   (random (length possible-genes)))))
           (if (equal? new-gene old-gene) ; got the same answer back
               (smart-mutate course-genes max-spread) ; try again
               (swap-gene course-genes zap-course new-gene 0 max-spread)))))))

;; swap-gene:
;;   Change one gene to another choice.
;;   Helper function for (smart-mutate). Attempts to recursively resolve
;;   conflicts by picking new genes. Returns #f if it can't do so within the
;;   maximum depth.
;;  course-genes: the original course genes vector
;;  course-num:         the index of the course to be modified
;;  new-gene:     the gene pair to replaced the current one
;;  depth:              the current number of gene propagations
;;  max-depth:          maximum number of gene propagations
(define (swap-gene course-genes course-num new-gene depth max-depth)
  (if (> depth max-depth)
      #f                                    ; done
      (let ((old-gene (vector-ref course-genes course-num))
            (new-course-genes (vector-copy course-genes)))
        (vector-set! new-course-genes course-num new-gene)
        (let ((affected
               ;; find a conflicting assignment (same ta)
               (vector-search
                    course-genes
                    (lambda (a)
                      (eq? (gpair-ta a) (gpair-ta new-gene))))))
          (if (not affected)
              new-course-genes                  ; done
              ;; try to fix the conflict
              (let* ((affected-index (vector-index-of course-genes affected))
                     (affected-others (vector-ref *alphabet* affected-index)))
                (if (= 1 (length affected-others))
                    #f
                    (swap-gene new-course-genes affected-index
                               (random-new-member affected-others affected)
                               (1+ depth) max-depth))))))))
```