Precept Topics

- Randomness
- Multiplicative Weights
- Decision Stumps and Boosting

A. RECAP: Randomness and Multiplicative Weights

Your preceptor will give an overview of the content of this week's lectures.

Feel free to use this space for notes or as scratch paper.

B. EXERCISE: Decision Stumps

In this problem, we will work through a small example of the *weak learner* you will be required to implement in the final programming assignment.

A *decision stump* is a very simple kind of binary classifier for points in k-dimensional space. (In the following examples, the dimension is $k = 2$.) Its decision depends on three values:

- the *dimension predictor* d_p, an integer between 0 and $k - 1$;
- the *value predictor* v_p, a real number; and
- the *sign predictor* $s_p \in \{0, 1\}$.

With these three values, the decision stump outputs a prediction for the *label* (i.e., either 0 or 1) of a sample point $x = (x_0, x_1, \ldots, x_{k-1})$ as follows:

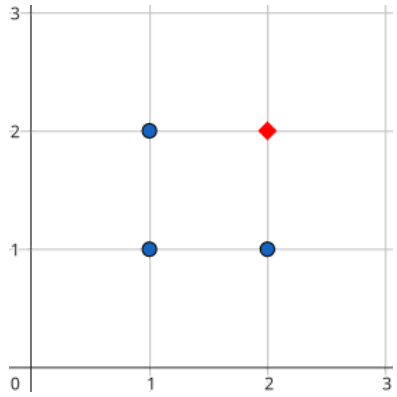
- if $s_p = 0$, output 0 if $x_{d_p} \leq v_p$ (and output 1 if $x_{d_p} > v_p$);
- if $s_p = 1$, output 1 if $x_{d_p} \leq v_p$ (and output 0 if $x_{d_p} > v_p$).

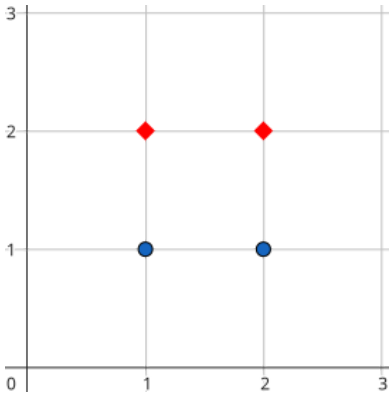
For example, if $d_p = 1$, $v_p = 0$ and $s_p = 1$, the predicted labels of $x = (0, 0)$, $y = (100, -2)$ and $z = (-100, 1)$ are 1, 1 and 0, respectively.

a) In the dataset examples below, count the number of correctly classified points (i.e., points whose predicted label matches the actual label) for decision stumps with the following values:

1. $d_p = 0$, $v_p = \frac{1}{2}$ and $s_p = 0$;
2. $d_p = 0$, $v_p = \frac{1}{2}$ and $s_p = 1$;
3. $d_p = 0$, $v_p = 1$ and $s_p = 0$;
4. $d_p = 0$, $v_p = 2$ and $s_p = 0$;
5. $d_p = 1$, $v_p = 1$ and $s_p = 0$;
6. $d_p = 1$, $v_p = 2$ and $s_p = 0$.

(Blue circles denote points labeled 0 and red diamonds denote points labeled 1. Dimension 0 corresponds to coordinates in the horizontal axis, while dimension 1 corresponds to the vertical axis.)





b) You worked out how to compute the classification of a decision stump given the dimension, value and sign predictor. But how should we *train* a decision stump, i.e., pick values for d_p, v_p and s_p?

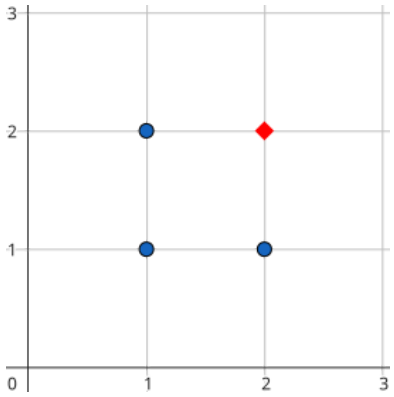
The algorithm is quite simple to state (if not so simple to implement): pick *optimal* values, i.e., those that minimize the number of incorrect predictions. One of the main challenges in your assignment will be to find these values efficiently.

What are the optimal values of d_p, v_p and s_p for the two datasets shown in part a)? Assume that the only allowed choices for v_p are values at some coordinate of some point in the dataset.

(If there is a tie between two sets of values – i.e., they misclassify the same number of points – we break ties down by dimension, then by value, then by sign. In other words, we pick the values where d_p is smaller; if they're equal, then we pick the smallest v_p; and if both d_p and v_p are equal, we pick the smallest s_p.)

C. EXERCISE: Boosting

Unfortunately, no decision stump can classify all points correctly in the first dataset of the previous problem (shown again below). How about using many stumps instead?



Boosting is a technique that enables us to increase the accuracy of a weak learner (like a decision stump). We create several decision stumps, penalizing misclassified points by doubling their weight (and thus the cost of their misclassification). More precisely, we first assign a weight of $1/4$ to each point in the dataset; then, a boosting iteration:

- creates a new decision stump for the dataset with the current weights;
- double the weights of misclassified points; and
- renormalize the weights (i.e., divide them by a number so that they sum to 1 again).

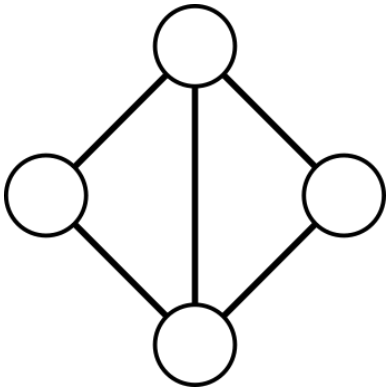
Each decision stump now chooses d_p, v_p and s_p to minimize the *weight* (rather than number) of misclassified points. The boosted classifier outputs the majority answer of all weak learners (i.e., if more than half of decision stumps predict 0, then so does the boosted classifier; and likewise for 1). For simplicity, assume the boosted learner makes an odd number of iterations, so there's no need for tie-breaking.

How many iterations are required for the boosted classifier to correctly classify *all* points in the dataset above? What are the values of d_p, v_p and s_p for the corresponding decision stumps?

D. EXERCISE: Random Spanning Trees

In this problem, we'll see how to generate a *uniformly random spanning tree* in a graph. We'll consider a few natural ideas and see how they behave on a small graph. (Maze generation is one cool application of random spanning trees, but they even show up in quantum field theory!)

a) First, let's consider a small example where we'll test our candidate algorithms:



Draw *all* spanning trees of the graph above. What are the probabilities of these trees under the uniform distribution?

b) Consider the following algorithm for sampling spanning trees from a graph: start with a tree containing a single (arbitrary) vertex. At each iteration,

1. choose an edge uniformly at random from the edges with exactly one endpoint in the tree;
2. add the edge to the tree; and
3. output the tree once there are no more edges to add.

Show that this algorithm does *not* generate uniformly random spanning trees. *Hint: show that there is a spanning tree that can only be obtained if its edges are sampled in one particular order.*

c) Consider the following alternative:

1. apply a uniformly random permutation to the edges;
2. run Kruskal's algorithm (without sorting – add edges in the order given by the permutation); and
3. output the resulting spanning tree.

Show that this algorithm does *not* generate uniformly random spanning trees. *Hint: pick one spanning tree and count how many permutations make the algorithm not select the missing edges.*

Note. *Aldous-Broder* and *Wilson's* algorithms are two examples of algorithms that sample uniform spanning trees; but they use *random walks*, where the algorithm may get stuck revisiting nodes in a partial tree over and over again. (The probability they do do for t steps decreases exponentially with t – they're Las Vegas algorithms, guaranteed to output a uniformly random spanning tree but whose runtime is a random variable.) This makes the analysis more difficult – we need to consider an infinite number of steps – but the algorithms have been proven to work.

Bottom line: sampling spanning trees is (also) hard!

EXERCISE (optional): Quicksort Analysis

Note: parts a) and b) of this exercise were the optional problem in Precept 4, but c) and d) are new.

You have seen, in lecture, that quicksort has $\Theta(n^2)$ worst-case runtime – but this is very (very!) unlikely to happen if we run `StdRandom.shuffle()` beforehand. Let's calculate how unlikely that really is.

Let `a` be an array of length n. Recall that the worst case occurs when `StdRandom.shuffle(a)` turns out to sort the array (either in increasing or decreasing order). Assume in this problem that `StdRandom.shuffle()` generates a uniformly random permutation and that all keys are distinct. (Feel free to use a calculator.)

a) What is the probability `StdRandom.shuffle(a)` sorts `a` when $n = 10$? How about $n = 100$?
How do these compare against the number of nanoseconds since the Big Bang?
Do the same with any other unfathomably large number of your choice.

b) Now let's try our hand at a probabilistic argument that shows quicksort usually makes significant progress. Call a pivot choice *bad* if it divides a subarray very unevenly: less than 10% of its size on one side, and more than 90% on the other. Call the pivot *good* otherwise.

What is the probability that the first 10 pivot choices are all bad? How about the probability that the first 100 choices are bad?

c) If the initial array has size n and at least one of the first 100 pivots is good, what is the largest size of a subarray defined by the pivots?

d) (Challenging) Try to use this bound to argue that quicksort runs in $\Theta(n \log n)$ time except with tiny probability. What problem do we run into? Does the argument work assuming n isn't too big?