Problem Set 5

General

This is the last problem set in 6.034!

To work on this problem set, you will need to get the code.

- You will need to **download and install** an additional software package called Orange for the second part of the lab. In the past, Orange has not worked with Python 2.6. The new download appears to now require Python 2.6. Please download Orange first so that you get the problems worked out early. If you have downloaded and installed it, you should be able to run `orange_for_6034.py` and get the Orange version number. Once you've filled in the boosting part, you should be able to run `lab5.py`, and see the output of several classifiers on the vampire dataset. If you get errors, email us.

- Orange is available for Linux (Ubuntu), Windows, and OS X (and we have successfully tested this lab on these platforms). Also, if you are working on Mac OS X, be sure that you have Numpy installed.

- For Ubuntu users: Please follow the install instruction here (see section on Linux). Note: just running `apt-get orange` on Ubuntu will *NOT* install orange but a completely different software package!

- ADVICE: If you can't get orange to work directly on your machine, try working on *Athena instead. It may not be a wise use of your time trying to get the Orange software to work on your machine, as it is only needed for half of lab 5. If orange works right out of the box great, but if it doesn't then follow the *Athena instructions to do the lab.

- To check that your Orange is properly installed, run:

  ```
  python orange_for_6034.py
  ```

  and you should get a version string and no errors.

- The built in Python 2.5 seems to work okay with Orange on Linux.

To work on this lab on *Athena you'll need to:

1. If you use bash: Setup your environment to pick up the class installation of Orange by adding the following:

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

```
export
LD_LIBRARY_PATH=/afs/athena.mit.edu/course/6/6.034/lib/python2.5/dist-
packages/orange:$LD_LIBRARY_PATH
```

to your .bashrc.mine file. If you are using tcsh or csh run:

```
setenv LD_LIBRARY_PATH
/afs/athena.mit.edu/course/6/6.034/lib/python2.5/dist-
packages/orange:$LD_LIBRARY_PATH
```

instead. You may omit `:$LD_LIBRARY_PATH` if you get complaints about it not being set. The above settings tell python where to look for shared libraries required for running Orange. NOTE: If you don't know what the shell you are running, type "echo $SHELL".

2. Log into linux.mit.edu: `ssh -X <user>`
3. For your convenience, we've provided a script, `run-orange-gui.sh` that will run the Orange GUI. Note that some Orange Widgets (like ROC charting) may not appear in GUI in the *Athena version. Don't worry, you won't need the GUI to answer the questions for this lab.

Your answers for the problem set belong in the main file `lab5.py`, as well as `neural_net.py` and `boost.py`.

Neural Nets

In this part of Lab 5, you are to complete the API for a Neural Net. Then afterwards, you are to construct various neural nets using the API to solve abstract learning problems.

Completing the implementation

We have provided you a skeleton of the Neural Net in `neural_net.py`, you are to complete the unimplemented methods.

The three classes, `Input`, `PerformanceElem`, and `Neuron` all have incomplete implementation of the following two functions:

```
def output(self)
def dOutDX(self,elem)
```

`output(self)` produces the output of each of these elements.

`dOutDx(self, elem)` generates the value of the partial derivative, given a weight element. Your first task is to fill in all 6 functions to complete the API.

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

The output functions for these three elements are the same as defined in class. Recall that the output for a neuron is computed using the sigmoid decision function:

```
s(z) = 1.0 / (1.0 + e**(-z))
```

Where z is the sum of the products of it's weights and inputs.

The output of a performance function is similarly standard:

```
p = -0.5 * (d - o) ** 2
```

Now on to the derivatives:

Recall, neural nets update a given weight by computing the partial derivative of the performance function with respect to that weight. The formula we have used in class is as follows:

```
weight[i] = weight[i] + rate * dP / dweight[i]
```

In our code this is represented as (see `def train()`)

```
w.set_next_value( w.get_value() + rate * network.performance.dOutdX(w) )
```

The element passed to the dOutdX function is always a weight. Namely it is the weight that we are doing the partial over. Your job is to figure out how to define dOutdX() in terms of recursively calling dOutdX() or output() over the inputs + weights of a network element.

For example, consider the Performance element P, `P.dOutdX(w)` could be defined in the following recursive fashion:

```
dP/d(w) = dP/do  * do/dw     # applying the chain rule
        = (d-o)  * o.dOutdX(w)
```

Here o, is the output of the Neuron that is directly connected to P.

For Neuron units, there would be more cases to consider. Namely, 1) The termination case where the weight being differentiated over is one of the (direct) weights of the current neuron. 2) The recursive case where the weight is not one that is directly connected (but is a descendant weight).

This implementation models the process of computing the chain of derivatives recursively. This top-down approach works from the output layer towards the input layers. This is in contrast to the more conventional approach (taught in class) where you computed a per-node little-delta, and then recursively computed the weight updates bottom-up, from input towards output layers.

If you are confused about how the top-down recursive chaining of derivatives work, first review the neural net notes. If you are still confused, ask the TAs for hints and clarifications.

Be sure to use the sigmoid and ds/dz functions as discussed in class

```
o = s(z) = 1.0 / (1.0 + e**(-z))
ds(o)/dz = s(z) * (1 - s(z))
```

The performance function and its derivative as discussed in class

```
P(o) = -0.5 (d - o)**2
dP(o)/dx = (d - o)
```

About the API Classes

Most of the classes in the Neural Network inherit from the following two abstract classes:

`ValuedElement`

This interface class allows an element to have a settable value. `Input` (e.g. i1, i2) and `Weight` (e.g. w1A, wAB) are subclasses of `ValueElement`

Elements that are subclassed all have these methods:

- `set_value(self,val)` - set the value of the element
- `get_value(self)` - get the value of the element
- `get_name(self)` - get the name of the element

`DifferentiableElement`

This abstract class defines the interface for elements that have outputs and are involved in partial derivatives.

- `output(self)`: returns the output of this element
- `dOutdX(self,elem)`: returns the partial derivative with respect to another element

`Inputs`, `Neurons`, and `PerformanceElem` are the three subclasses that implement `DifferentiableElements`. You will have to complete the interface for these classes.

`Weight(ValuedElement)`

Represents update-able weights in the network. It has in addition to ValueElement functions:

- `set_next_value(self,val)`: which sets the next weight value in self.next_value
- `update(self)`: which sets the current weight to the value stored in self.next_value

Both methods are used for the training algorithm itself and you will not need them in your implementation.

`Input(DifferentiableElement, ValuedElement)`

Represents inputs to the network. These may represent variable inputs as well as fixed inputs (i.e. threshold inputs) that are always set to -1. `output()` of Input units should simply return the value they are set to during training or testing.

`dOutdX(self, elem)` of an Input unit should return 0, since there are no weights connect directly into inputs.

`Neuron(DifferentiableElement)`

Represents the actual neurons in the neural net. The definitions for output and dOutdX already contains some code. Namely, we've implemented a value caching mechanism to speed up the training / testing process. So instead of implementing `output` and `dOutdx` directly you should instead implement:

- `compute_output(self):`
- `compute_doutdx(self,elem):`

`PerformanceElem(DifferentiableElement)`

Represents a Performance Element that allows you to set the desired output.

- `set_desired` which sets the desired_output.

The desired output is set during training. You will not need it for your implementation.

To better understand back-propagation, you should take a look at the methods **`train`** and **`test`** in `neural_network.py` to see how everything is put together.

Unit Testing

Once you've completed the missing functions, we have provided a python script `neural_net_tester.py` to help you unit test. You will need to pass the tests in this unit tester before you can move on to the next parts.

To check if your implementation works, run:

```
python neural_net_tester.py simple
```

This makes sure that your code works and can learn basic functions such as AND and OR.

Building Neural Nets

Once you have finished implementing the Neural Net API, you will be tasked to build three networks to learn various abstract data sets.

Here is an example of how to to construct a basic neural network:

```
def make_neural_net_basic():
    """Returns a 1 neuron network with 2 variable inputs, and 1 fixed
input."""
    i0 = Input('i0',-1.0) # this input is immutable
    i1 = Input('i1',0.0)
    i2 = Input('i2',0.0)

    w1A = Weight('w1A',1)
    w2A = Weight('w2A',1)
    wA  = Weight('wA', 1)

    # the inputs must be in the same order as their associated weights
    A = Neuron('A', [i1,i2,i0], [w1A,w2A,wA])
    P = PerformanceElem(A, 0.0)

    # Package all the components into a network
    # First list the PerformanceElem P
    # Then list all neurons afterwards
    net = Network(P,[A])

    return net
```

Naming conventions

IMPORTANT: Be sure to use the following naming convention when creating elements for your networks:

Inputs:

- Format: `'i' + input_number`
- Conventions:
    - Start numbering from 1.
    - Use the same `i0` for all the fixed -1 inputs
- Examples: `'i1'`, `i2`.

Weights:

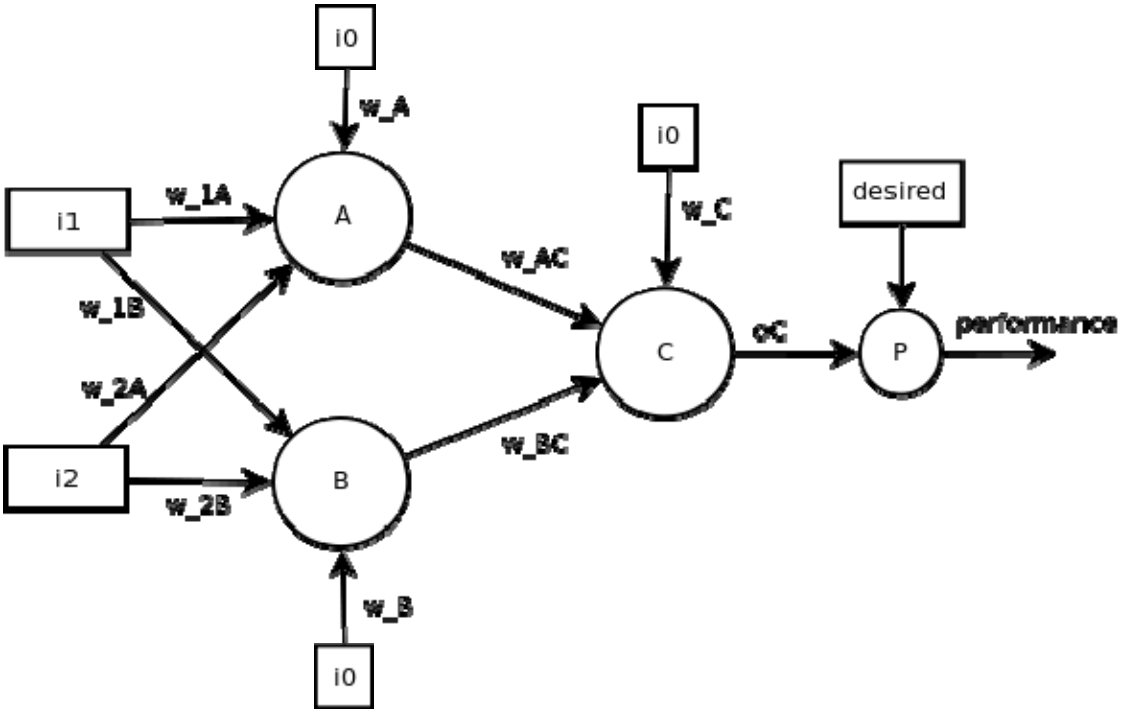
- Format `'w' + from_identifier + '.' + to_identifier`
- Examples:
    - `w1A` for weight from Input `i1` to Neuron `A`
    - `wBC` for weight from Neuron `B` to Neuron `C`.

Neurons:

- Format: `alphabet_letter`
- Convention: Assign neuron names in order of distance to the inputs.
- Example: `A` is the neuron closest to the inputs, and on the left-most (or top-most) side of the net.
- For ties, order neurons from left to right or top to bottom (depending on how you draw orient your network).

Building a 2-layer Neural Net

Now use the Neural Net API you've just completed and tested to create a two layer network that looks like the following.



Fill your answer in the function stub:
`def make_neural_net_two_layer`
in `neural_net.py`.

Your 2-layer neural net should now be able to learn slightly harder datasets, such as the classic non-linearly separable examples such as NOT-EQUAL (XOR), and EQUAL.

When initializing the weights of the network, you should use random weights. To get deterministic random initial weights so that tests are reproducible you should first seed the random number generator, and then generate the random weights.

```
seed_random()

wt = random_weight()
...use wt...
wt2 = random_weight()
...use wt2...
```

Note: the function `random_weight()` in neural_net.py uses the python function `random.randrange(-1,1)` to compute initial weights. This function generates values: -1, 0, 1 (randomly). While this may seem like a mistake but what we've found empirically is that this actually performs better than using random.uniform(-1, 1). Be our guest and play around with the

`random_weight` function. You'll find that Neural Nets can be quite sensitive to initialization weight settings. Recall what happens if you set all weights to the same value?

To test your completed network run:

```
python neural_net_tester.py two_layer
```

Your network should learn and classify all the datasets in the `neural_net_data.harder_data_set` with 100% accuracy.

Designing For More Challenging Datasets

Now it's your turn to design the network. We want to be able to classify more complex data sets.

Specifically we want you to design a new network that should theoretically be able to learn and classify the following datasets:

1. The letter-L.

```
4 + -
3 + -
2 + -
1 + - - - -
0 - + + + +
  0 1 2 3 4
```

2. This moat-like shape:

```
4 - - - - -
3 -       -
2 -   +   -
1 -       -
0 - - - - -
  0 1 2 3 4
```

3. This `patchy` shape:

```
4 - -   + +
3 - -   + +
2
1 + +   - -
0 + +   - -
  0 1 2 3 4
```

We claim that a network architecture containing 5 neuron nodes or less can fully learn and classify all three shapes. In fact we required it!

Construct a new network in:

```
def make_neural_net_challenging
```

that can (theoretically) perfectly learn and classify all three datasets.

To test your network on the first 2 of these shapes, run

```
python neural_net_tester.py challenging
```

To pass our tests, your network must get 100% accuracy within 10000 iterations.

Now try your architecture on the third dataset `patchy`. Run:

```
python neural_net_tester.py patchy
```

Depending on your architecture and your initial weights, your network may either easily learn `patchy` or get stuck in a local maxima. Does your network completely learn the dataset within 10000 iterations? If not, take look at the weights output at the end of the 10000 iterations. Plot the weights in terms of a linear function on a 2D graph. Does the boundaries tell you why there might be a local maxima?

Manually Setting Weights

You can have your network learn the dataset `patchy` perfectly and very quickly if the proper weights are set.

You can use either of these strategies to determine the proper weights.

1. You can experimentally determine the right weights by running your network until it perfectly learns the dataset. You will probably need to increase the max-iterations parameter, or playing around with different initial weight settings.
2. You can try to solve for the weights analytically. This is a good chance to put into practice the methods for solving network weights you learned in tutorial.
3. In either case, we want you to find preset weights to the same network that you built in the last part. Your new weight-preset network should be able to learn the `patchy` problem with only 1000 additional iterations of training.

After you've found the optimal weights, fill in:

```
def make_neural_net_with_weights
```

To test, run:

```
python neural_net_tester.py weights
```

If everything tests with an accuracy of 1.0. Then you've completed the Neural Networks portion of lab5. Congrats!

Now on to Boosting!

Boosting

You're still trying to use AI to predict the votes of politicians. ID-Trees were great, but you've heard about these other magnificent learning algorithms like SVMs and Boosting. Boosting sounds easier to implement and had a pretty good reputation, so you decide to start with that.

To make sure that you interpret the results without letting your political preconceptions get in the way, you dig up some old data to work with: in particular, the data from the 4th House of Representatives, which met from 1796 to 1797. (According to the records on voteview.com, this is when the two-party system first emerged, with the two parties being designated "Federalists" and "Republicans".)

You experiment with that data before going on to the 2007-2008 data, finding that Congressmen in 1796 were much more clear about what they were voting on than in 2008.

The framework for a boosting classifier can be found in `boost.py`. You need to finish coding it, and then use it to learn some classifiers and answer a few questions.

The following resources will be helpful:

- The documentation for the boosting code, which you can find embedded in `boost.py` in the documentation strings.
- The Shapire paper on boosting, or the notes that summarize it.
- The Lab 4 writeup, if you need to refer to how `data_reader.py` represents legislators and votes.

A (clever|cheap) trick

The boosting code uses a trick that means it only has to try half the number of base classifiers.

It turns out that AdaBoost does not really care which side of its base classifier is +1 and which side is -1. If you choose a classifier that is the *opposite* of the best classifier -- it returns -1 for most points that should be +1, and returns +1 for most points that should be -1, and therefore has a high error rate -- it works the same as if you had chosen the negation of that classifier, which is the best classifier.

If the data reweighting step is implemented correctly, it will produce the same weights given a classifier or its opposite. Also, a classifier with a high error rate will end up with a *negative* alpha value, so that in the final "vote" of classifiers it will act like its opposite. So the important thing about a classifier isn't that its error rate is *low* -- it's that the error rate is *far from 1/2*.

In the boosting code, we take advantage of this. We include only classifiers that output +1 for voting YES on a particular motion, or +1 for voting NO on a particular motion, and as the "best classifier" we choose the classifier whose error rate is *farthest from 1/2*. If the error rate is high, then the result will act like a classifier that outputs +1 for "voting NO or abstaining", or +1 for

"voting YES or abstaining", respectively. This means we don't have to include these classifiers in the base classifier list, speeding up the algorithm by a factor of 2.

Completing the code

Here are the parts that you need to complete:

- In the `BoostClassifier` class in `boost.py`, the `update_weights` method is undefined. You need to define this method so that it changes the data weights in the way prescribed by the AdaBoost algorithm. There are two ways of implementing this update which happen to be mathematically equivalent.
- In the `BoostClassifier` class, the `classify` method is also undefined. Define it so that you can use a trained BoostClassifier as a classifier, outputting +1 or -1 based on the weighted results of its base classifiers. Complete the very similar `orange_classify` method as well.
- In `lab5.py`, the `most_misclassified` function is undefined. You will need to define it to answer the questions.

Remember to use the supplied `legislator_info(datum)` to output your list of the most-misclassified data points!

Questions

Answer the two questions `republican_newspaper_vote` and `republican_sunset_vote` in `lab5.py`.

When you are asked how a particular political party would vote on a particular motion, disregard the possibility of abstaining. If your classifier results indicate that the party *wouldn't* vote NO, consider that an indication that the party would vote YES.

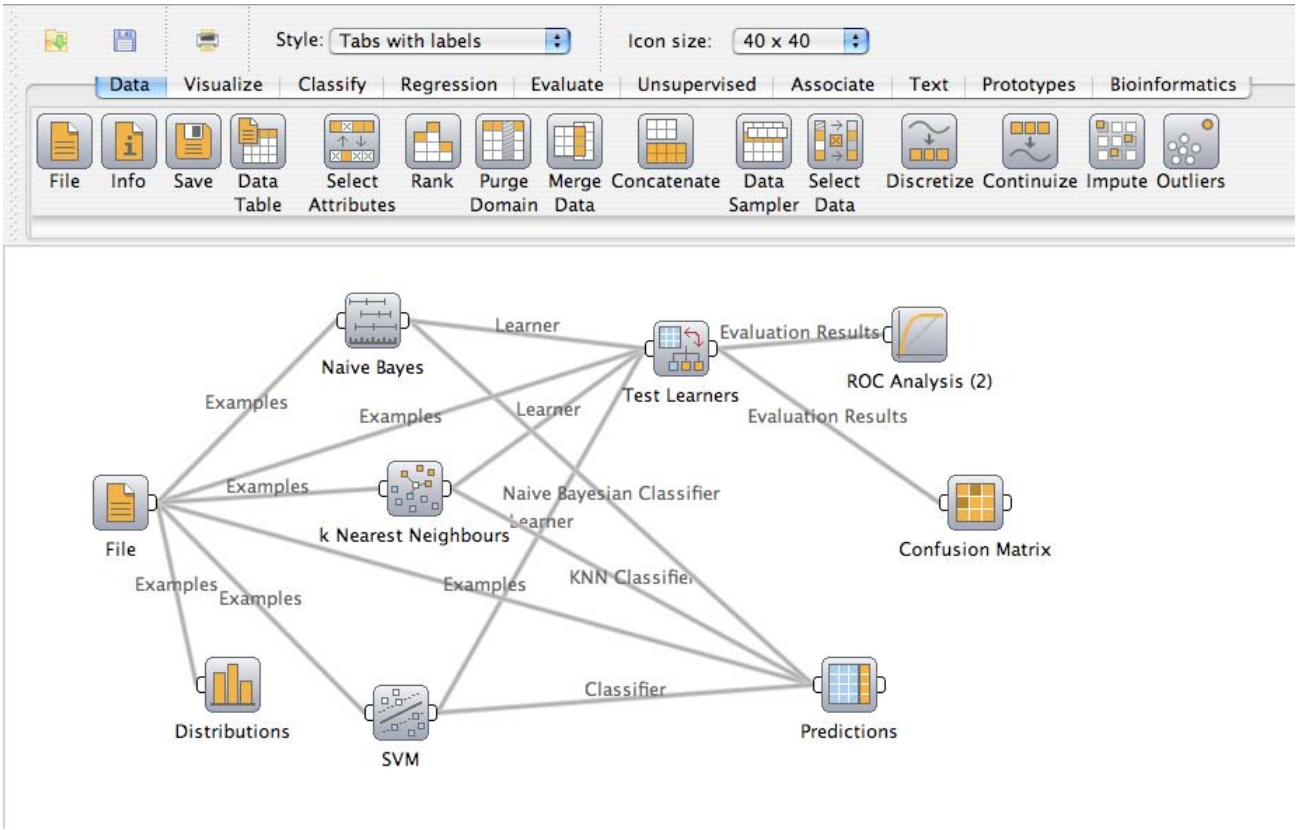
Orange you glad someone else implemented these?

First things first: Have you installed Orange yet?

Now that you've installed Orange, when you run lab5.py, does it complain about Orange, or does it show you the outputs of some classifiers on vampire data?

Getting familiar with the Orange GUI

This part is to get you familiar with the Orange GUI to do a little machine learning without doing any programming. We've given you some data files (vampires.tab, H004.tab, adult.tab, titanic.tab, breast-cancer.tab, etc.) that you can play with. Try making something like the following:



Then take a look at the performance, and look at the actual predictions.

Using Orange from Python

We have given you a function called `describe_and_classify` that trains a bunch of classifiers that Orange provides. Some of them will be more familiar than others.

First it trains each classifier on the data, shows its output on each data point from the training data, and shows the accuracy on the training data. You know from class that the accuracy on the training data should be 1 for these classifiers. It is less than one because each classifier comes with built-in regularization to help prevent overtraining. That's what the pruning is for the decision tree, for example. We didn't specify regularization parameters to most of the learners because they have fine defaults. You can read more about them in the Orange documentation.

You'll notice that we do one extra thing with the decision tree. We print it out. For most classifiers, their internal representations are hard for humans to read, but the decision tree's internal representation can be very useful in getting to know your data.

Then `describe_and_classify` passes the untrained learners, without having seen the data, to cross-validation. Cross-validation hides part of the training data, trains the learner on the rest, and then tests it on the hidden part. To avoid accidentally picking just one terrible subset of the data to hide (an irreproducible subset), it divides the data evenly into some number of folds,

successively tests by hiding each fold in turn, and averages over the results. You will find with cross-validation that the classifiers do much better on identifying political parties than they do on vampires. That's because the vampire dataset has so few examples, with so little redundancy, that if you take away one example, it is very likely to remove the information you actually need in order to classify that example.

To ensure that you've gotten the right idea from each part of `describe_and_classify`, there are **six questions** just below the function.

Boosting with Orange

You may be able to do better by using AdaBoost over your Orange classifiers than you could do with any of those Orange classifiers by themselves. Then again, you may do worse. That AdaBoost "doesn't tend to overfit" is a somewhat conditional statement that depends a lot on the particular classifiers that boosting gets its pick of. If some of the underlying classifiers tend to overfit, then boosting will greedily choose them first, and it will be stuck with those choices for testing.

We have set up a learner that uses the BoostClassifier from the first part, but its underlying classifiers are the Orange classifiers we were just looking at. When you combine classifiers in this way, you create what's called an "ensemble classifier". You will notice, as you run this new classifier on the various data sets we've provided, that the ensemble frequently performs worse in cross-validation than some (or most) of its underlying classifiers.

Your job is to find a set of classifiers for the ensemble that get **at least 74% accuracy on the breast-cancer dataset**. You may use any subset of the classifiers we've provided. Put the short names of your classifiers into the list `classifiers_for_best_ensemble`. There will be honorable mention in class for the best ensemble. Classifier performance appears to be architecture dependent, so you might be able to get to 74% with just one classifier on your machine, but that won't be enough on the server -- in this case, try to get even better performance at home. If you are proud of the way that you went about choosing the best ensemble, let us know to look at your code carefully, and there may be another honorable mention for that.

Bonus!

We think there may be a problem with the code somewhere, but we don't know quite where. If you run the ensemble boosting with the original set of learners (one of each) then you get terrible results for the H109 data set. Can you figure out why?

Errata

Neural Nets

If you are having problems with getting your network to convergence on certain problems, try the following:

1. Change your random weight function to use random.randrange(-1,1) instead of random.uniform(-1,1). We've found that the former produced weights that are more conducive to network convergence.
2. Order your weight initialization (i.e. calls to random_weight()) from the bottom-most weights to the top-most weights in you network. While this ordering theoretically irrelevant, we've found that this ordering worked well in practice (in conjunction with 1 above). NN are unfortunately quite sensitive to initial weight settings.
3. Play around with the seed_random function to try different starting random seeds. Though seeding the random function with 0 is what worked for us.
4. If none of these work, try setting weights that are close to what you want in terms of the final expected solution.

Boosting and Orange

It turns out that different architectures yield different results from `describe_and_classify` on the vampire and H004 datasets. Some folks have perfectly reasonable answers to the short answer part given the output on their machine, but do not pass the tests with those answers.

For the same reason, some folks are getting 74% on their home computers very easily. The question was intended to set a lower limit which was larger than any of the classifiers by itself. If you're getting 74, but the ensemble classifier test is failing, you'll need to try harder. You'll want to use some subset of the classifiers. Think about why the decision tree isn't a good choice as a base classifier for boosting.

6.034 Artificial Intelligence

Fall 2010