Philosophy & Physical Computing: Lab Manual

Benjamin C. Jantzen

June 18, 2019

Contents

Chapter 1

Measuring the atmosphere

1.1 Initialization procedure

1. If your RPi is on, shut it down.

> To power off the RPi, click on the root menu and select "Shutdown". There are two indicator lights on the side of the RPi, close to where the power cord plugs in. One is red and will be lit whenever the RPi is connected to power. The other is green. When you shutdown the RPi, it is important that you DO NOT CUT OFF POWER until the green light flashes, glows solid, and then turns off. Once the green light is off (and the red light glows steady), you can press the button on the power cord to shut off power.

2. While off, disconnect the power cord to the RPi and connect it instead to the battery. The RPi will boot up. Once connected to the battery, you can velcro the RPi to the top.
3. Log on.
4. Open a terminal window.
5. type: `cd code/utilities`
6. Press [Enter]
7. type: `python ThermoMeasure.py`
8. Press [Enter]

9. Press the button on the RPi. The LEDs should flash in sequence, and the screen should display the message: `Button pressed!`
10. Press CTRL+C and you should see a bunch of numbers appear on the screen.
11. type: `python ThermoMeasure.py`
12. Press [Enter]
13. This time, while the program is running, turn off the monitor and then carefully unplug the keyboard, the mouse, the ethernet cable, and the monitor. Do *not* press the button.
14. Carry the RPi and attached battery outside, and proceed to the next section.

1.2 Measurement procedure

1. Carry your apparatus to Hahn Garden.
2. Once in the garden, hold the apparatus in a variety of locations – in the open sun, in the shade, near the pond, low to the ground, etc. – and press the button to take a handful of samples in each location.
    - Give the sensors a few seconds to equilibrate in each new location before pressing the button.
    - Let the LEDs finish flashing before pressing the button again.
    - Be sure to take more than one sample at each spot.
    - Try to be consistent with regard to how you hold the apparatus, whether you are blocking the wind, etc.
    - Aim for variety and quantity.

1.3 Reconnection and data processing

1. Once you're done sampling, return to the lab.
2. Be careful not to press the sampling button once indoors.
3. While leaving the battery connected to the power terminal of the RPi, carefully reconnect the keyboard, mouse, ethernet, and monitor cables. Turn the monitor on.

4. Press CTRL+C.
5. This terminates the sampling program; if successful, a bunch of numbers should appear on the screen.
6. type: `python ProcessThermoData.py`
7. Press [Enter]
8. A bunch of graphs should appear on your screen. The instructor will explain their significance.

Chapter 2

A very brief introduction to Python

Note: If you are already familiar with Python and numpy, please move directly to Chapter 3 and begin programming your own perceptron.

2.1 Overview

Before we do anything else, let's settle with tradition. Open a terminal window, type `ipython`, and then press [Enter]. This launches a Python interpreter in interactive mode; you can give it Python commands and it will carry them out. In this case, type the following on a single line, and then press [Enter]:

```
print "Hello, world!"
```

You should see a greeting appear on the next line. Congratulations! You just wrote and executed your first Python program. Now type `quit` and hit return. Launch *Geany* (it's under the **Programming** category in the home menu), and type the same line of code into the open document. Save the file as "hello_world.py". Now exit *Geany*. In the terminal window, type the following:

```
python hello_world.py
```

Press [Enter] and that familiar greeting should appear on the next line. You've now executed Python code in two different ways: as commands entered directly to the interpreter and as a script launched from the command line!

With tradition out of the way, let's take a more organized approach to a Python crash course. We're going to make use of the excellent online tutorials at Code Academy: Python. So close everything else and click on the link in the preceding sentence to open a web browser. You may want to resize your windows so you can read these instructions and the tutorial at the same time. Since we don't have time to present a complete course in Python, we're going to be a little selective.

2.2 Syntax, variables, and dynamical typing

1. Under "Unit 1" on the Codecademy page, click on "Lesson: Python Syntax". Follow the instructions to complete the unit. (Note: ignore messages of `None` that appear at the bottom of the output. At times, you'll have to click on "continue to the next section". Unfortunately, Codecademy has changed policy and now requires that you either sign in with an existing account or create a new one. If you prefer not to create an account with your own credentials, the instructor will be happy to supply some.

   There are 13 sections to Unit 1. Once you get to the "Tip Calculator" exercise (after the "Review" section), just click the "L̂earn Python 2" link at the top of the page to see the list of Units again.

2. Once you've finished this first unit, open *Geany* and write three lines of code to do the following:

   (a) Assign x a value of 2^{3^4}

   (b) Make a variable, y, to store the square root of the floating point value 3.14159.

   (c) Assign a variable z to 12 mod 10

3. Add the following three lines to your program:
   ```
   print x
   print y
   print z
   ```

4. Save your program in your home folder (`/home/ppc/`)

5. Open a terminal window and run your script (type `python your_script_name.py`).

6. You should get three lines of output:
   ```
   241785163922925834941235 2L
   1.7724531023414978
   2
   ```

2.3 Strings

1. Under "UNIT 2: STRINGS AND CONSOLE OUTPUT", complete the "Lesson: Strings & Console Output".

This lesson is less relevant to what we'll be doing, but it's important to know how to use the `print` command.

2.4 Control flow

1. Under "UNIT 3: CONDITIONALS AND CONTROL FLOW", complete "Lesson: Conditionals & Control Flow". There are 15 sections. This unit is important; flow control is part of nearly every nontrivial program.

2. Once you've completed the unit, test your knowledge by writing a program that takes a number as input and prints out 0 if and only if that number is even. You can write and execute your program in *Geany*.

2.5 Functions

1. Under "UNIT 4: FUNCTIONS", complete "Lesson: Functions". There are 19 sections.

2. Once you've completed the unit, write a function that accepts a number and, if it's even, squares it, or if it's odd cubes it, and returns the result. Include the definition of the function in a script that also calls that function for the argument and then prints the output. That way, you can both write and test your function in *Geany*.

2.6 Lists

1. Under "UNIT 5: LISTS & DICTIONARIES", complete the first 9 sections; you can skip the material on dictionaries for now.

2.7 Lists and functions

1. Under "UNIT 7: Lists and Functions," complete "Lesson: Lists and functions." There are 18 sections.

2.8 Loops

1. Under "UNIT 8: Loops", complete "Lesson: Loops". There are 19 sections.

2. After you've completed this unit, write a function that checks whether two lists are of the same length and, if so, computes their dot product. In other words, for lists $[x_1, x_2, \ldots, x_n]$ and $[y_1, y_2, \ldots, y_n]$ compute $x_1y_1 + x_2y_2 + \cdots + x_ny_n$.

2.9 Classes

1. Under "UNIT 11: INTRODUCTION TO CLASSES," complete "Lesson: Introduction to Classes." There are 18 sections. We'll practice making classes by building a perceptron later.

2.10 Numpy

2.10.1 Documentation

Numpy is a package (actually a subset of Scipy) containing many classes, methods, etc. useful for numerical and scientific computation. Like Python, it is free and open-source. You can find much detailed documentation and source code online.

2.10.2 Arrays

1. Read through the section entitled "The Basics" in the 'Quickstart tutorial.' To enter the example code as you go along and view the results, open a terminal window and type `ipython --pylab`. This

will launch an interactive Python interpreter that already as numpy loaded. You do *not* need to enter the lines that say

```
import numpy as np
```

2. To test your knowledge, try creating two arrays of length 100 containing linearly spaced values, that is, values that are evenly spaced over an interval (hint: see `np.arange` and `np.linspace`). Multiply the arrays element by element to generate a third array of length 100. Then, create a fourth array that contains only the last 10 entries of the third.

3. Compute the dot product of two arrays (of the same length) you define. Note how much easier this is with numpy than when we used Python loops.

Chapter 3

Programming a Perceptron[1]

3.1 Getting started

1. Launch *Geany* (it's under the **Programming** category in the home menu).
2. From the **File** menu, choose **Save As...** Name the file "MyPerceptron.py", and chose the "code" folder, and then click **Save**.

3.1.1 What do I put at the beginning?

Since we'll want to work with vectors (ordered lists of numbers that can be added and scaled and for which an inner product is defined), we first want to load **numpy**, a package of numerical methods for Python. To have this package loaded whenever your own Perceptron module is loaded, enter the following at the top of your program:

```
import numpy as np
```

This form of the command let's us rename the imported module to the more concise **np** for easy reference in our own program. Since Python uses indentation as part of its syntax (rather than, say, brackets), it's important to *not* indent this first line.

[1] We'll be following the structure of the class defined in Raschka (2015, p25-26).

3.2 Writing code

3.2.1 Define a class

The first thing to do is to define a class (see Code Academy: Python, Unit 11). This will allow us to bundle together important data (such as the value of c) and important 'methods' or routines (such as training the perceptron with given data) that will be associated with particular instances of the perceptron classifier (referred to as objects). In other words, a class defines a type of thing – in this case a perceptron classifier – and provides a handy way to associate all of the stuff needed to set up and use the classifier with each instance we create. This way, we can use the same class to create as many classifiers with different properties as we like. Here's the shell of a class definition which you can enter underneath your **import** command (skip a line or two first):

```
class Perceptron(object):

    def __init__(self, c=0.01, n_iter=10):
                ...
```

Note that you shouldn't actually type the dots above; they're there just to remind you that you need to fill in the guts of the function. The

```
__init__
```

function lets you set some values or perform other initialization operations when you mint a fresh instance of the class. In this case, it sets the learning rate (which we've been calling c as well as the number times the algorithm should loop over the given data. It would also be sensible to initialize the set of weights, w_i, as a vector of zeros. So, replace the dots above with 3 lines making these variable assignments. Note that class attributes that aren't meant to be seen or used directly by the user are conventionally named with a leading underscore. Here's a template to get you started (replace the '?' symbols):

```
class Perceptron(object):

    def __init__(self, c=0.01, n_iter=10):
        self.c = ?
        self.n_iter = ?
        self._w = ?
```

Hint: For creating a vector of weights, $\vec{w}$, we want to create a numpy array. One way to do so is to create a Python list of zeros of the right length and then use the method

```
np.array()
```

on that list. But an easier method can be found here.

3.2.2 Add methods

Now we'd like to associate with our class some 'methods' or procedures for taking action. In particular, we need to provide an implementation of the learning algorithm. Recall that the algorithm goes like this:

1. Choose a value for the constant, $c > 0$ (which controls strength of corrections or rate of learning). It's usually 1.
2. Choose initial values of the weights (usually 0).
3. Repeatedly cycle through the training examples (in random order). For each example considered that has value $t \in -1, 1$ adjust, the wights according to the equation

$$\Delta w_j = c(t - a)x_j \tag{3.1}$$

Note that this algorithm requires us to know a, the classification predicted by the perceptron given its current weights. To keep things simple, lets split that computation into two parts: (i) weighting each of the input values and summing them up; and (ii) comparing the result to the threshold. Let's also include a special weight w_0 that always receives 1 as an input (so we can learn boundaries that don't pass through the origin).

1. So we want to introduce a method that takes a vector $\vec{x}$ (a numpy array) as input and computes the following:

$$\vec{w} \cdot \vec{x} = \sum_{i=1}^{n} w_i x_i + w_0 \tag{3.2}$$

   Try writing such a method (name it `net_input`). Remember that a classes' methods must take 'self' as an input. Again, you can do this using a loop and Python lists. But it can be done in a single line if you use this numpy method: `np.dot`.

Tip: If you use numpy methods to write your method, then it should be possible to pass in a matrix, essentially a column of vectors with each row representing a point in feature space. Using the `np.dot`, you'll get back a vector containing the value of the dot product for each point. To see how it behaves, run the following code:

```
x = [[1, 2], [3, 4], [5, 6]]
y = [7, 8]

np.dot(x,y)
```

The output should be:

```
array([23, 53, 83])
```

2. Once you've got the `net_input` method working, you can add another method that calls it in order to return the overall prediction of the perceptron. That is, given an input vector, this second method calls `net_input` on that vector, and then compares the result to 0.0. If it's greater than or equal to zero, the method should return a 1 for that data point, otherwise -1. If you're slick about it, you can use the `np.where` method. This will work on all rows at once if you've passed all data in as a single matrix.

3. Now for the meat of the learning algorithm. Write a function with the following declaration:

```
def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
```

```
        self : object

        """
```

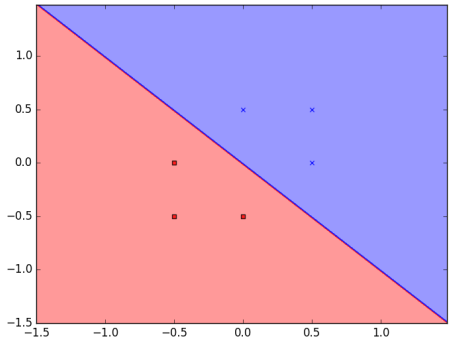
The comments explain what the inputs are supposed to be. Now fill in code that computes an update to the weight vectors and then applies that update. Remember that your updates depend upon predictions. These are computed by the method you completed above. You'll need a parameter to determine how many times to loop through the training data. You can pass that in. A good default value is 10. If you wish, you can similarly include a parameter, c that controls the learning rate by scaling the correction factor for each correction to the weight vector.

3.2.3 Test your code

1. Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

```
$ cd code
$ ipython --pylab
: from MyPerceptron import *
: X = np.array([[-0.5, -0.5], [-0.5, 0.], [0., -0.5], [0.5, 0.5], [0.5, 0.], [0.,
: y = [-1, -1, -1, 1, 1, 1]
: p = Perceptron()
: p.fit(X, y)
: from utilities.PlotDecisionRegions import *
: plot_decision_regions(X, y, p)
```

2. If all goes well, you should see something close to the following:



3.3 Advanced

If you have completed your own perceptron and want to tackle some more advanced tasks, try the following:

1. Make your perceptron an "online" learner that can be updated continuously by passing in new data.

2. Make your perceptron stable even when the data is *not* linearly separable. In other words, make it so that your perceptron halts on some sort of sensible answer even when you cannot draw a line through your data that separates the classes.

Chapter 4

SVM and kernel methods

4.1 Loading your data

- Navigate to the `code/utilities` directory.
- Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

  ```
  $ ipython --pylab
  : from HumidityClassification import *
  ```

- If you'd like to use your own data from the garden, enter the following:
  `X_train, y_train, X_test, y_test = hlt_prep()`
- If you'd prefer to use the large data set taken by the instructor, enter this instead:
  `X_train, y_train, X_test, y_test = hlt_prep(source='tesseract')`

4.2 Support Vector Machines (SVMs)

The previous commands loaded our data in a format useful for training some classifiers and testing their performance. In this section, we'll see how well a Support Vector Machine (SVM) performs. In particular, we'll use a linear SVM to classify our data. To do so, enter the following commands:

```
: from sklearn import svm
: clf_svm = svm.SVC(kernel='linear')
```

```
: clf_svm.fit(X_train, y_train)

: from PlotDecisionRegions import *

: plot_decision_regions(X_train, y_train, clf_svm)
```

To see how well your trained classifier does, enter:

```
: clf_svm.score(X_test, y_test)
```

Now try a nonlinear SVM (one that uses a nonlinear 'kernel'). Enter the following:

```
: clf_nlsvm = svm.SVC(kernel='rbf')

: clf_nlsvm.fit(X_train, y_train)

: plot_decision_regions(X_train, y_train, clf_nlsvm)

: clf_nlsvm.score(X_test, y_test)
```

4.3 Nearest Neighbor Classifier

Now let's see how the Nearest Neighbor Classifier performs. Enter the following commands in the ipython session:

```
: from sklearn.neighbors import KNeighborsClassifier

: clf_knn = KNeighborsClassifier(1)

: clf_knn.fit(X_train, y_train)

: plot_decision_regions(X_train, y_train, clf_knn)

: clf_knn.score(X_test, y_test)
```

Chapter 5

Regression and function estimation

5.1 Loading your data

1. First, load your data:

   - Navigate to the `code/utilities` directory.
   - Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

     ```
     $ ipython --pylab
     : from HumidityRegression import *
     ```

   - If you'd like to use your own data from the garden, enter the following:
     `X_train, y_train, X_test, y_test = hlt_prep()`
   - If you'd prefer to use the large data set taken by the instructor, enter this instead:
     `X_train, y_train, X_test, y_test = hlt_prep(source='tesseract')`

2. Next, to make things easy to visualize, let's just look at a single feature, namely temperature, and estimate humidity:

   - `T_train = X_train[:,0].reshape(len(X_train),1)`
   - `T_test = X_test[:,0].reshape(len(X_test),1)`
   - `from sklearn import linear_model`

- `clfLinear = linear_model.LinearRegression()`
- `clfLinear.fit(T_train, y_train)`
- `clfLinear.score(T_test, y_test)`
- `plot(T_train, y_train, 'bo', T_train, clf.predict(T_train))`

3. Repeat this series of commands for a 'lasso' regression model (using `clfLasso = linear_model.Lasso()`) and a 'ridge regressor' (using `clf = linear_model.Ridge()`).

4. Repeat your experiments using the full `X_train` set (not just temperature). Note that when you do so, you probably don't want to run the plot commands (they won't make much sense).

Chapter 6

Ohm's Law

In this experiment, you're going to be exploring the elecctrical properties of wires and batteris (or in this case, a potentiometer and power supply).

6.1 Building your experimental apparatus

The first thing to do is to use the wires, components and breadboard provided to build the following circuit you see in Figure 6.1. *Do not connect your circuit to the power supply until it's been checked by an instructor.*

To connect the wires labeled "Analog 0", "Analog 1", and "Analog ground" to the DAQ board on top of your Raspberry Pi, refer to the diagrams in Figure 6.1 and 6.1 below. You'll need to borrow a small screwdriver to loosen the screws, insert the end of a wire, and then tighten them again.

Once your circuit is completed and connected to the DAQC board, please ask an instructor to verify that it is correct before adding the connections to the power supply.

6.2 Gathering data

1. Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

   ```
   $ cd code/utilities
   $ python MeasureCurrent.py
   ```

2. Once the program launches, follow the instructions on the screen.

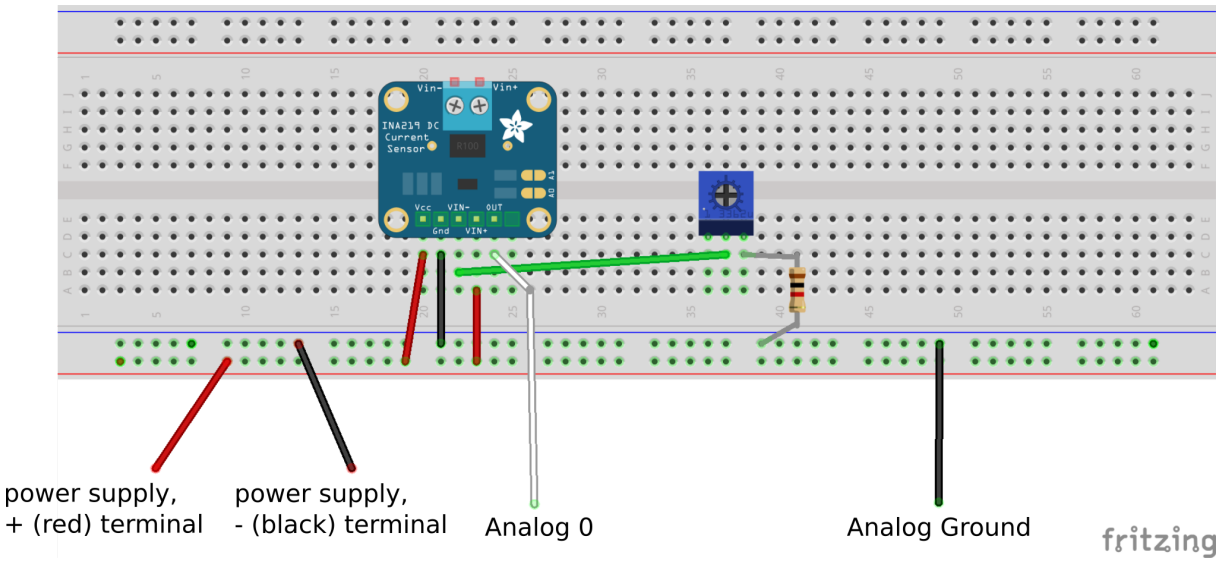


Figure 6.1: The circuit you should build.

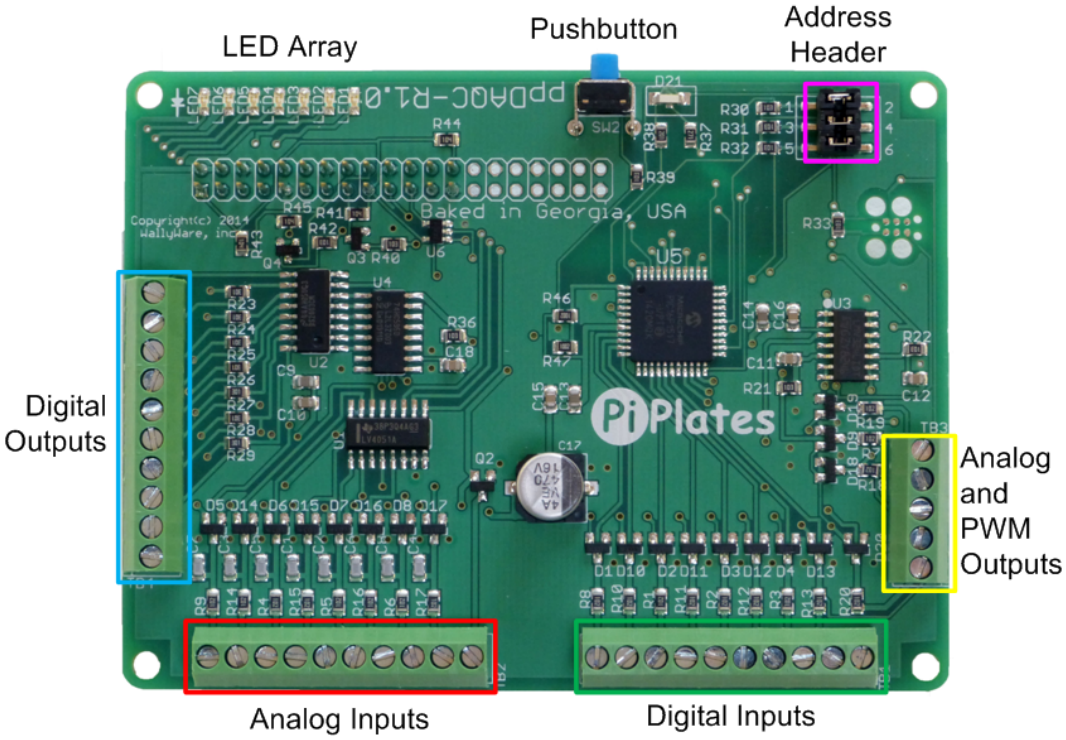


Figure 6.2: The parts of the DAQC board.

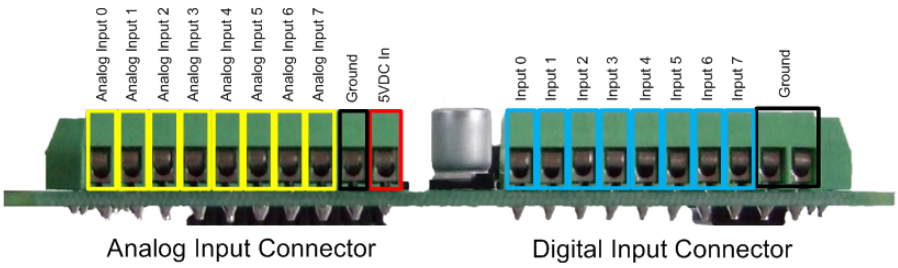


Figure 6.3: The analog inputs.

Chapter 7

Causal analysis

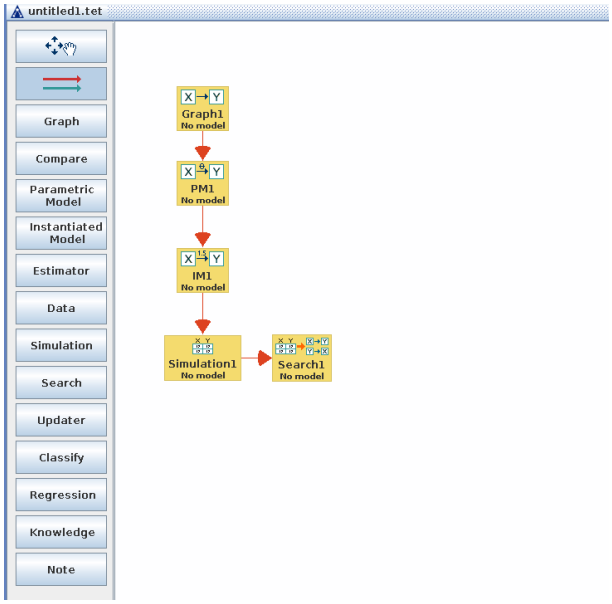
7.1 Trying out inference in Tetrad

1. Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

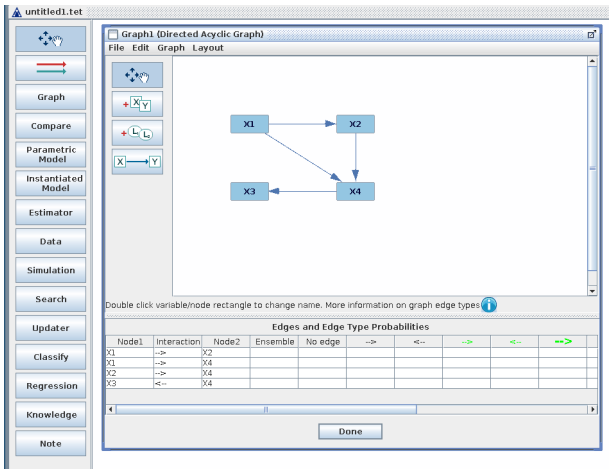
   `$ cd code`

   `$ java -jar tetrad-gui-6.6.0-launch.jar`

2. Be patient; it takes time for Tetrad to open. Once it does, you should see a blank workspace with a column of buttons on the left and a menu bar at the top. From the menu, choose `Pipelines` → `Simulate from a given graph, then search` The result should look like this:



3. Double-click the box labeled "Graph". Choose "Directed Acyclic Graph" from the list of options and click "OK". A window then opens in which you can use the buttons to the left to draw a graph. Use these to create a DAG of at least four variables and containing no latents (only use variables surrounded by rectangles). Here's an example:

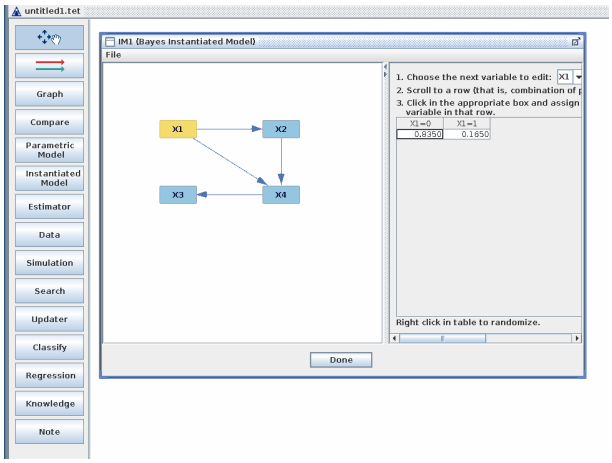


When you're satisfied with your graph, click the "Done" button at the bottom.

4. Double-click the box labeled "PM1". This box is for building a para-

metric model on top of your DAG. Specifically, it lets you decide what sort of mathematical model will be used and how many parameters of each type it requires. In this case, we'll use a Bayes Net. So choose that option, and when a window pops up offering options under the heading: "Categories for variables should be:" select "Automatically assigned". Then click "OK". Another window then opens, showing you the details of the model you've just created. When you're done looking around, click "Done".

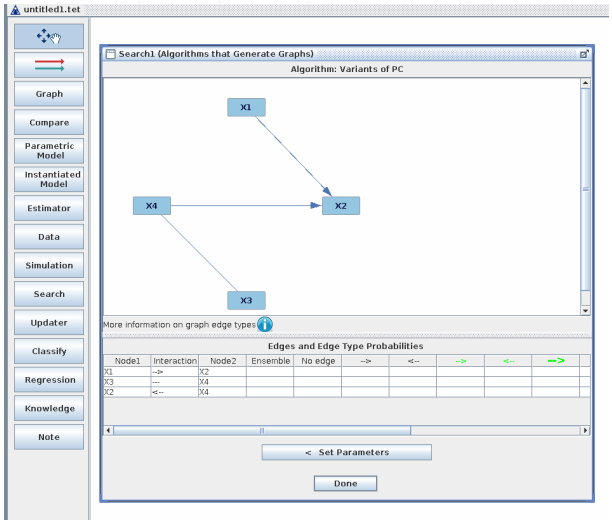
5. Now double-click the box "IM1" to make a specific instantiation of your Bayes Net model. This will assign specific values to the parameters in your model, in this case to the different probabilities making up the joint distribution. Click "OK" in the window that opens, and then choose "Randomly" in the "IM Structure Editor" that opens.A window showing your instantiated model then opens. It looks like this:



When you're done looking around, click "Done".

6. The Simulation box is going to let us generate data as if it came from a system described by your instantiated model. To get it to do so, double-click the "Simulation1" box. In the window that opens, click "Done".

7. Now to try to learn the true graph from the simulated data we generated. Double click the "Search1" box. In the window that opens, choose the "Variants of PC" option from the list of algorithms that can be selected, and then click the "Set Parameters" button at the bottom of the window. Then click "Run Search and Generate Graph". An output graph will then appear. Here's an example output:



7.2 Learning from your own data

1. Open a terminal and execute the following commands:

   ```
   $ cd code/utilities
   $ python ThermoData2CSV.py
   ```

2. In Tetrad, open a new workspace.

3. Add a Data node to your workspace.

4. Double click on the "Data1" node. Select `File` → `Load Data` and choose the file that begins with "atmo_data" and ends in ".csv" from your `data/output` folder.

5. Now add a Search node and double-click it. I suggest selecting the FGES algorithm to run on your data, but you are welcome to explore others.

Chapter 8

BACON

8.1 Trying out heuristic search

1. Open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

   ```
   $ cd code/completed_examples
   $ ipython --pylab
   ```

2. Once the python interpreter launches, we're going to import the BACON module and use it to search for laws given Boyle's original data on the temperature and pressure of gases. Enter the following:

   ```
   : from BACON import *
   : table_boyle = Table()
   : table_boyle.load_data("../../data/boyle.pkl")
   : c = Constant()
   : l = Linaer()
   : i = Increasing()
   : d = Decreasing()
   : table_boyle.find_laws([c,l,i,d])
   ```

8.2 Trying out variable invention

1. If you still have open the ipython session in which you defined the `table_boyle` object in the preceding section, then you can skip right

to entering the code in Step 2 below. Otherwise, open a terminal and enter the following commands (don't type the '$' or ':' symbols – they just indicate the terminal or ipython prompt, respectively):

```
$ cd code/completed_examples
$ ipython --pylab
```

2. Once the python interpreter launches, we're going to import the BACON module and use it to search for laws and (possibly) invent variables given your data on the current passing through various wires. Enter the following:

```
: from BACON import *
: table_ohm = Table()
: table_ohm.load_data("../../data/output/ohm_law_data_COMPUTER_NAME.pkl")
```

   Note: `COMPUTER NAME` is the name of your particular RaspberryPi. The quickest way to enter the file name you need is tab-completion. Just type the command up to `ohm_law_data_` and hit TAB. ipython will fill in the rest for you.

```
: c = Constant()
: l = Linaer()
: i = Increasing()
: d = Decreasing()
: table_ohm.find_laws([c,l,i,d])
```

   If everything goes as expected, the last command will return an empty set. You can check this by entering `table_ohm._laws`. It should output `set()`. This is because BACON, as we implemented it, can only find functional relations between quantitative variables. But two of your variables are qualitative (describing the distinct resistances and "batteries" used. But BACON can do more. Try the following:

```
: table_ohm.invent_variables()
```

   This time, a bunch of data will (likely) be printed to the screen. Type `table_ohm._terms`. If you the list that prints has more than three entries, then BACON has invented it's own intrinsic variable and added it to the three with which you began! To check that this is the case, enter: `table_ohm._terms[3]._definition`. If all went well, it will say `'intrinsic1'`.

Bibliography

Raschka, Sebastian (2015). *Python Machine Learning*. English. Packt Publishing - ebooks Account.