G54MRT

Python and Sensors 4:
Testing and Tuning Algorithms

Joe Marshall, Stuart Reeves

Functional Prototype Versus Final System

- In practice, often have multiple prototypes for a system – what you're building for coursework is a functional prototype
- Does the bare bones of **what** a final system does
- Underlying logic is the same as final system
- Not the same form factor
  - Note: can also do form factor prototype without prototyping actual function (see Introduction to HCI course for more about these)
- Not the same outputs
  - Can also prototype outputs with fake input / sensor data
  - e.g. Wizard of Oz methods

What are you testing

- Whether (part of) your system does what it is supposed to do.
- Break down testing into parts, e.g.:
  - Sensor characteristics, filtering etc.
  - Functional behaviour / underlying logic
  - Whole system testing
  - Live/in-situ testing?

Testing Versus Algorithm Tuning

- You'll find yourself testing things each time you do some code.
- You may need to repeatedly tune and re-run aspects of your algorithm (thresholds etc.), this is a type of continuous testing.
- Continuous tuning / testing is great, and should be done.
  - Allows you to have an idea of how well your system is working.
  - Make sure you haven't broken something
- You also want to do systematic tests once you have built the system
  - Allow you to know (and demonstrate to us) what the performance of the system is
  - Check unusual cases you may not have considered in development
- This lecture talks about both things
  - You will often want to do similar things during each phase
  - But it may be appropriate to use different data-sets

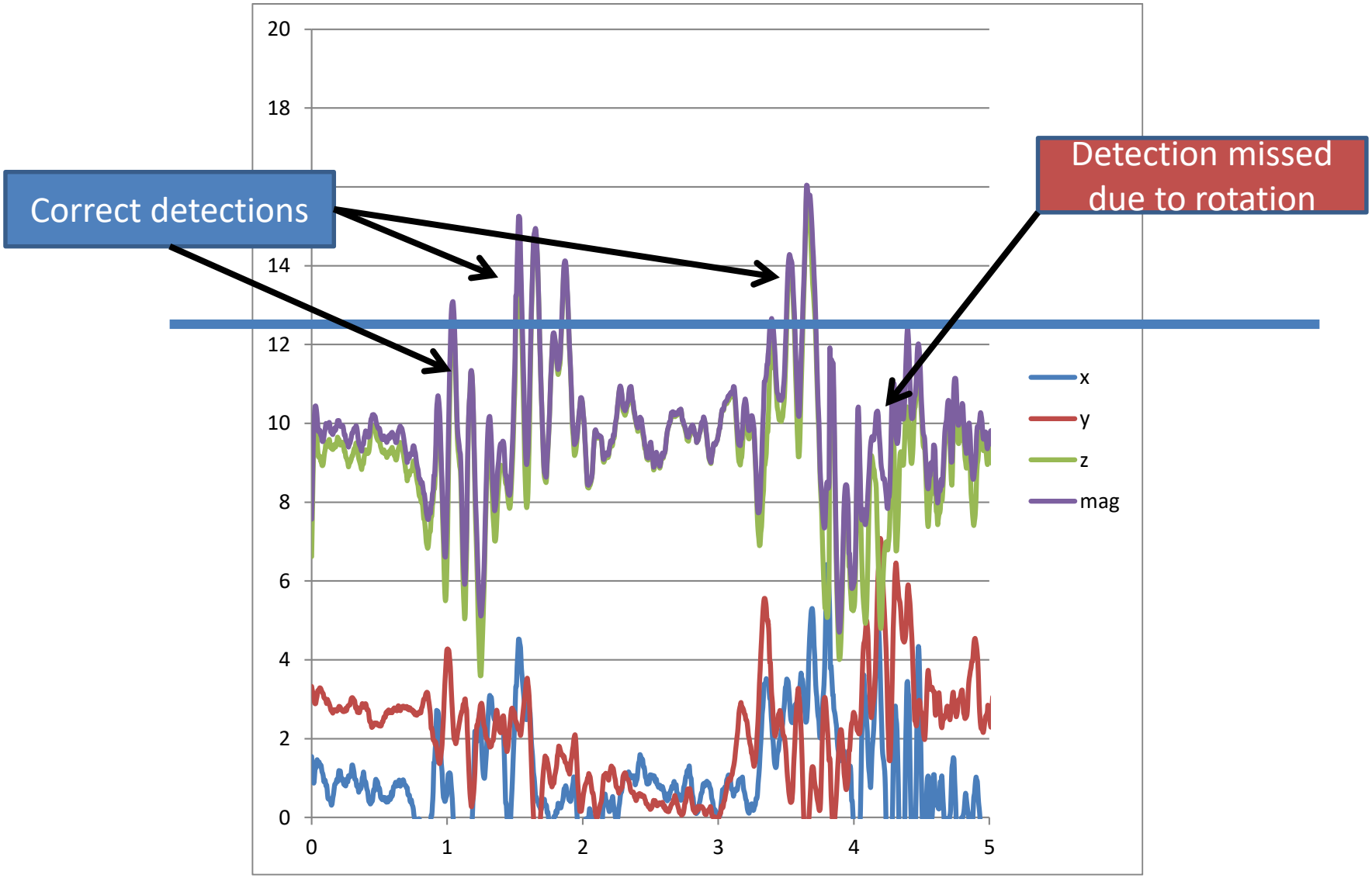
Documenting testing

- Always take a note of:
  - What tests you did
  - What was your test environment / setup
    - Possibly photos, sketches or diagrams of physical setup
  - What data-set you ran it on
  - What were the results of the test
    - Graphs or statistics if applicable
- Some of this detail may not end up in your report
  - But you can't know what is important until you finish.
- Suggestions here for statistics to collect, types of graphs to use
  - Dependent on the details of your algorithm what is appropriate

Testing sensor characteristics and filtering

- How do the sensors work?
  - E.g. How often does the motion sensor fire if someone is sat in a room. How often does it fire if no-one is in front of it?
- If you filter to detect an event on a sensor, is your filtering detecting this event correctly?
  - **What percentage of events are correctly detected?** (True positives)
  - **How often does it identify an event when there isn't one?** (False positives)
  - Tuning thresholds to do this
- If you filter to detect some aspect of the sensed situation, does your filtering work
  - E.g. Does your sound level sensing respond in a way that fits with some kind of ground truth, e.g.
    - **Mean squared error based on some other sound level metering**
    - **Fit with your perception of sound levels**
- What causes errors?
  - In depth look at data (e.g. graph examples of false/true positives)
  - Look at what was happening when the error occurred (e.g. what you were doing)

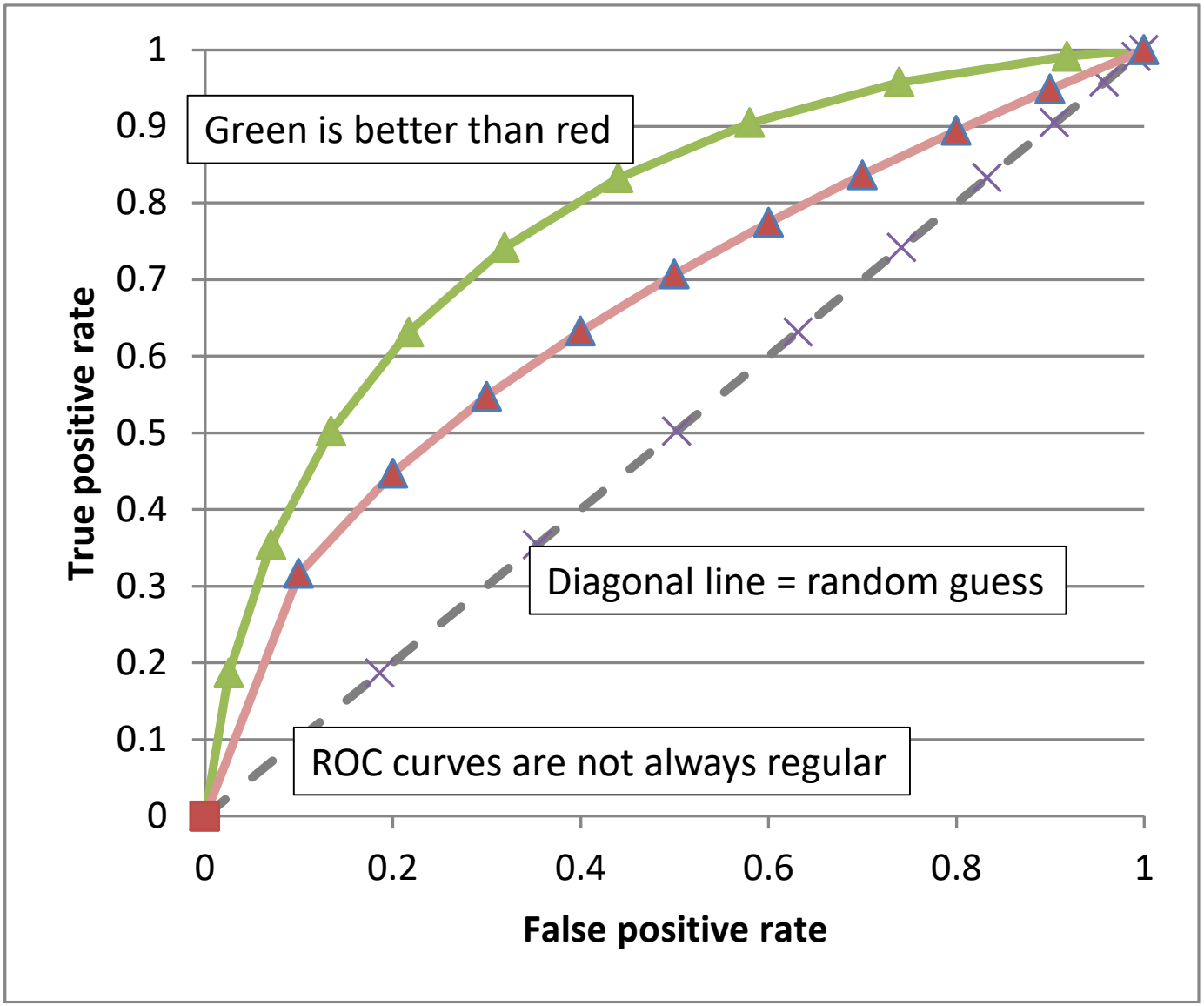
Shake detection threshold example



Receiver Operating Characteristic (ROC) Curves

- If you are making a binary decision, you need to balance false positives versus false negatives.
- How you balance it may depend on the context of your system
  - What user impact will there be of missing an event?
  - What impact will a false alarm have?
- You do this by changing some threshold or parameter in your algorithm, the **discrimination threshold.**
- If you systematically vary your discrimination threshold, you can get to a sweet spot, where you balance false positives and negatives for your system.
- ROC curves are a way of displaying how the balance between cases occurs in your system (and what point you've chosen)
- Also useful in demonstrating how good your system is – how much better than random chance.
- Plot false positives against false negatives.
  - Because it is independent of algorithm details
  - Can annotate showing algorithm threshold values chosen for your optimum spot
- Area under ROC curve is often used as a measure of relative algorithm quality
  - A way of choosing between two algorithms, independent of threshold

ROC Curve Example



Measurements of numerical error

- You are tracking some kind of numerical value
- You have ground truth for that value.
- e.g.

Number of people in lab (real)	Algorithm estimate
1	2
2	2
5	3
7	10
9	11

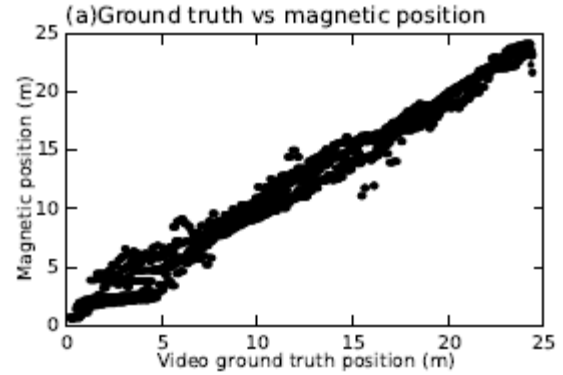
- You can use basic descriptive statistics to test and demonstrate how good your algorithm is.

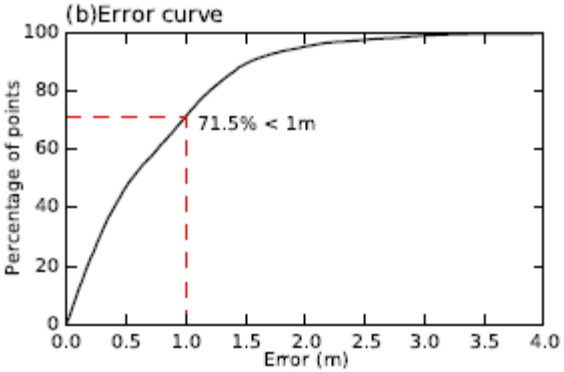
Measurements of numerical error

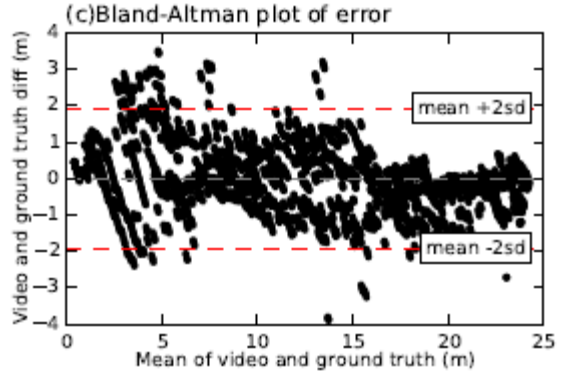
- Point error: $\mathrm{error(t)} = (real(t) - algorithm(t))$
  - How wrong is each individual data point
- Mean error: $meanerror = \sum_{t=0}^{n} \frac{error(t)}{n}$
  - How is the data centred relative to reality (bias)
- Error variance: $\frac{1}{n-1}\sum\left(error(t) - meanerror(t)\right)^2$
  - How much does the error vary
- Mean squared error: $\frac{1}{n}\sum error(t)^2$
  - Bigger if a lot of points are far away from reality.
  - Very widely used, e.g. when optimising algorithms it is common to minimise mean squared error.

Other things that you might do

- Plot ground truth against reality
- Error curves – show percentage of points which are within a certain error
- Bland-Altman plots
  - Visual way of showing how well real and ground truth values correlate.
- Correlation statistics
  - But beware, things can correlate well, but not actually be very good, use other error statistics too.







Logic testing

- Reduced / changed sensors
- Test harnesses
- About checking the logical tests that are at the core of your algorithm.
- Allows checking of edge cases / hard to check cases
  - E.g. 2 motion sensors (or anything motion sensory)
- Doesn’t test the sensor itself (see previous slides)
- Essentially software testing
- Emulator is useful for this kind of testing

Reduced/changed sensors

- Can test core algorithm logic with different sensors
- Example, monitoring motion in two rooms
  - Can't be in two rooms at once
  - The room your computer is in will always report motion
  - Hard to trigger all possible combinations
  - Can't always set them up in real positions
  - A pain to have to wave in front of sensors all the time
  - Swap them for: buttons, switches (rotary dial in a digital input), touch sensor etc, and you can test the core algorithm logic (e.g. what does it do when both rooms are busy?)
- Example, monitoring sound level:
  - You want to check how your algorithm responds to sound level, due to room you're working in, you can't create desired sound levels?
  - Replace loudness sensor with rotary dial
  - You can answer questions such as 'what does the algorithm do if it is very quiet?' 'what does the algorithm do if there is a sudden spike in noise?'

Test harness (no sensors)

- Take your core logic, and put that in a separate python file.
- Run that logic on some data
  - Data you generated synthetically to explicitly check all potential situations.
  - Data you pre-recorded to a log file for various situations (the emulator does log file replay for this purpose)
- So you can test the logic alone
  - Without having to generate sensor data every time
  - To distinguish between errors caused by algorithm logic, and errors caused by sensor noise etc.
  - Most of the time, algorithm logic should function correctly

Synthetic data example (1)

- Detection of knock patterns on a door to unlock the door.
- Knock pattern detector takes accelerometer data, and converts it to patterns based on the gap between each knock, to give something like:
  - Long, Short, Long, Long, Short (for 6 knocks = LSLLS)
- Height detection detects how tall the person knocking is.
- Each person has a known knock code and height.

Synthetic data example (2)

- We want to test:
  - Door unlocks if person uses the correct knock code and height is close enough to the correct height.
  - Door doesn't unlock otherwise.
- Knock code detection and height detection have already been tested (when we tested sensor characteristics and filtering)
  - We know how many errors in the knock code sentence are likely (and have chosen how many errors we want to accept based on desired true vs false positive rates, ROC curves etc)
  - We know the range of error in the height detection, and have chosen what range of error we are willing to accept.

Synthetic data example (3)

- Synthetic dataset columns:
  - Detected height
  - Detected knock pattern
  - Expected result or output
- Generate a dataset with examples of:
  - Correct knock pattern and height
  - Knock pattern and height with small errors inserted into each (within the acceptance threshold).
  - Wrong knock pattern, correct height
  - Wrong height, correct knock pattern
  - Everything wrong
- Run algorithm logic on this dataset and check it gives correct results for all cases
- This is just standard software testing - with a well written test harness, you can script all this, so any time you mess with your core algorithm logic, you can check it all tests okay.

Synthetic data example (4)

- Some algorithms rely on previous state, so you may also want to run examples back to back
  - E.g. Wrong knock pattern followed by correct knock pattern.

Example of a logic error

```
def setRedLED(onoff):
  digitalWrite(REDLEDPIN,onoff)

def setGreenLED(onoff):
  digitalWrite(GREENLEDPIN,onoff)

# red LED is on if the room is noisy and bright
if noiseLevel > 65 and lightLevel>500:
  setRedLED(1)
# green LED is on if room is not noisy and bright
if lightLevel<500 and noiseLevel<54:
  setGreenLED(1)
```

What is wrong with this algorithm?

Can you spot two errors?

Example of a logic error

```
def setRedGreenLEDs(red,green):
  digitalWrite(REDLEDPIN,red)
  digitalWrite(GREENLEDPIN,green)

# red LED is on if the room is noisy and bright
if noiseLevel > 65 and lightLevel>500:
  setRedGreenLEDs(1,0)
# green LED is on if room is not noisy and bright
if lightLevel<500 or noiseLevel<54:
  setRedGreenLEDs(0,1)
```

Using a single function makes sure that the LEDs aren't left on when they aren't meant to be

Whole system testing

- Test the system as a whole
- From sensors to outputs
- Either use a script, and perform the data, or use real data which you annotate
- Record everything
  - Raw sensor data
  - Processed sensor data (filtered, thresholded etc)
  - Internal state of the system?
  - Output from the system
- You may only need to record the raw sensor data once, then replay it when you fix things in an algorithm
  - Did I mention yet that the emulator allows you to replay data logs?
  - If the system relies on a feedback loop between sensor data and people's actions, you may need to run full tests each time.
- You can report results using methods described previously (false positive/negative rates, descriptive statistics, graphs of error rates etc.)

Testing script example

Time	Action	Expected result
30 seconds	Stand normally Knock LLLSL (correct knock)	Door opens, errors detected = 0
60 seconds	Bend right down, knock LLLSL	Door doesn't open, height warning message displayed.
90 seconds	Stand normally, knock LLLSS (1 error)	Door opens (with note that 1 error was detected)
120 seconds	Wear a hat, knock LLLSL	Door opens (with note that height off)
150 seconds	Wear a hat, knock LLLSS (1 error)	Door doesn't open (says height off and 1 error detected = too much error)

Live demo – people counting

```
lastPIR1Time=None
lastPIR2Time=None

lastNumberIn=0
numberIn=0
startTime=time.time()
print "time,pir1,pir2,numberIn"

# we only do things when the PIR values *rise*
lastPIR1=0
lastPIR2=0

while True:
    pir1=grovepi.digitalRead(PIR1_PIN)
    pir2=grovepi.digitalRead(PIR2_PIN)
    curTime=time.time()

    # is the incoming PIR value rising?
    if pir1 and lastPIR1==0:
        lastPIR1Time=curTime
        # did we recently see PIR 2? (within the last 2 seconds)
        # if so, lets assume someone came out, otherwise we assume it is someone coming in and
        # do nothing until we see PIR2
        if lastPIR2Time!=None and curTime-lastPIR2Time<2:
            numberIn-=1
    # is the outgoing PIR rising?
    if pir2 and lastPIR2==0:
        lastPIR2Time=curTime
        # did we recently see PIR 1? (within the last 2 seconds)
        # if so, lets assume someone came out, otherwise we assume it is someone coming in and
        # do nothing until we see PIR1
        if lastPIR1Time!=None and curTime-lastPIR1Time<2:
            numberIn+=1
    lastPIR1=pir1
    lastPIR2=pir2
    print "%f,%d,%d,%d"%(curTime-startTime,pir1,pir2,numberIn)
    time.sleep(0.1)
```

Real data (from start of lecture)

Live demo (crossed fingers)

Time	Event	Expected people in
10	1 person in	1
20	1 out	0
30	2 in	2
40	1 in and 1 out	2
50	2 out	0

Using real data example

- I’m measuring how busy the lab is (noisyness, number of people in the lab etc.)
- I can’t control this with a script.
- So I set sensor data recording, and sit in the lab for an hour, with a tested smartphone sound level meter app.
- Each minute I note down various things into a spreadsheet (recorded noise level, my perceived noisyness, number of people.)
- Once I’ve got this dataset I can see how my algorithm outputs compare
- Useful for tuning things that rely on interpretation – e.g. how does my perception of noise level (on a scale of 0-10) map to recorded noise level?

Real data example dataset

Time (mins)	Level Meter	My perceived noise level	Number of people
1	44	3	35
2	46	3	35
3	43	3	35
4	48	5	37
5	51	5	39
6	53	6	40
…	…	…	…

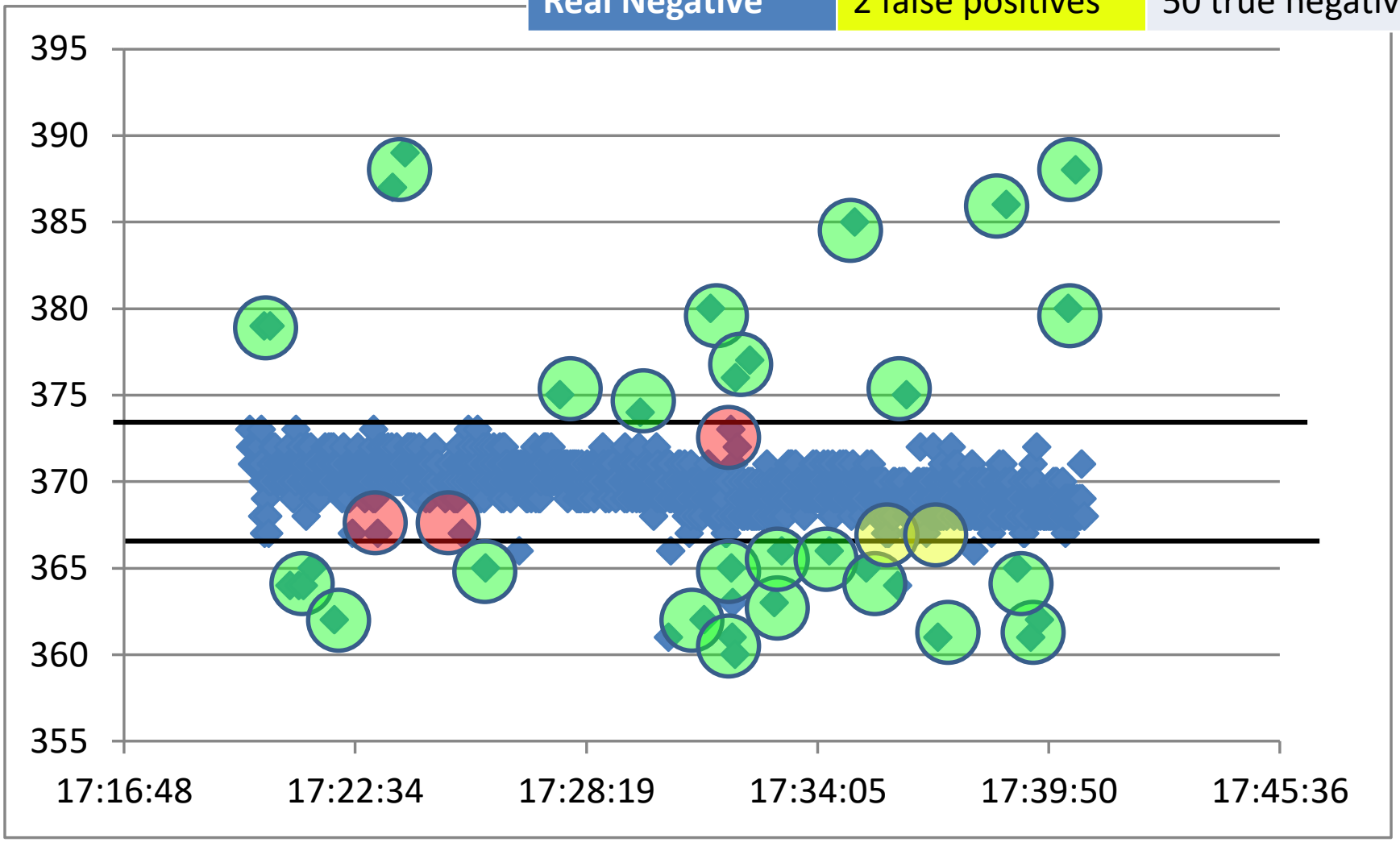
Tuning versus Testing Datasets

- Algorithm tuning (or algorithm training if you have automatic methods of doing it) will quite likely require some kind of pre-recorded dataset.
- To make the algorithm run as well as possible, you can run tests on this dataset, adjust thresholds, levels of filtering etc. and re-run the same tests.
- There is however a risk with this, of overtraining or overfitting
  - You tune algorithm to work with a specific dataset
  - But some aspects of the dataset are not inherent to the system, just values that happen to be that level for this recorded dataset.
- Typically, we record multiple datasets, and use one for tuning the algorithm during development (or training an automatic algorithm), and a second one for final testing to check algorithm performance.

Example of over-fitting

- Thresholding, trying to detect when someone passes or does something that shadows or reflects light onto the light sensor.
- So, we record data for 15 minutes, and detect movements past the sensor box.
- Then we build a thresholding algorithm based on this and evaluate it

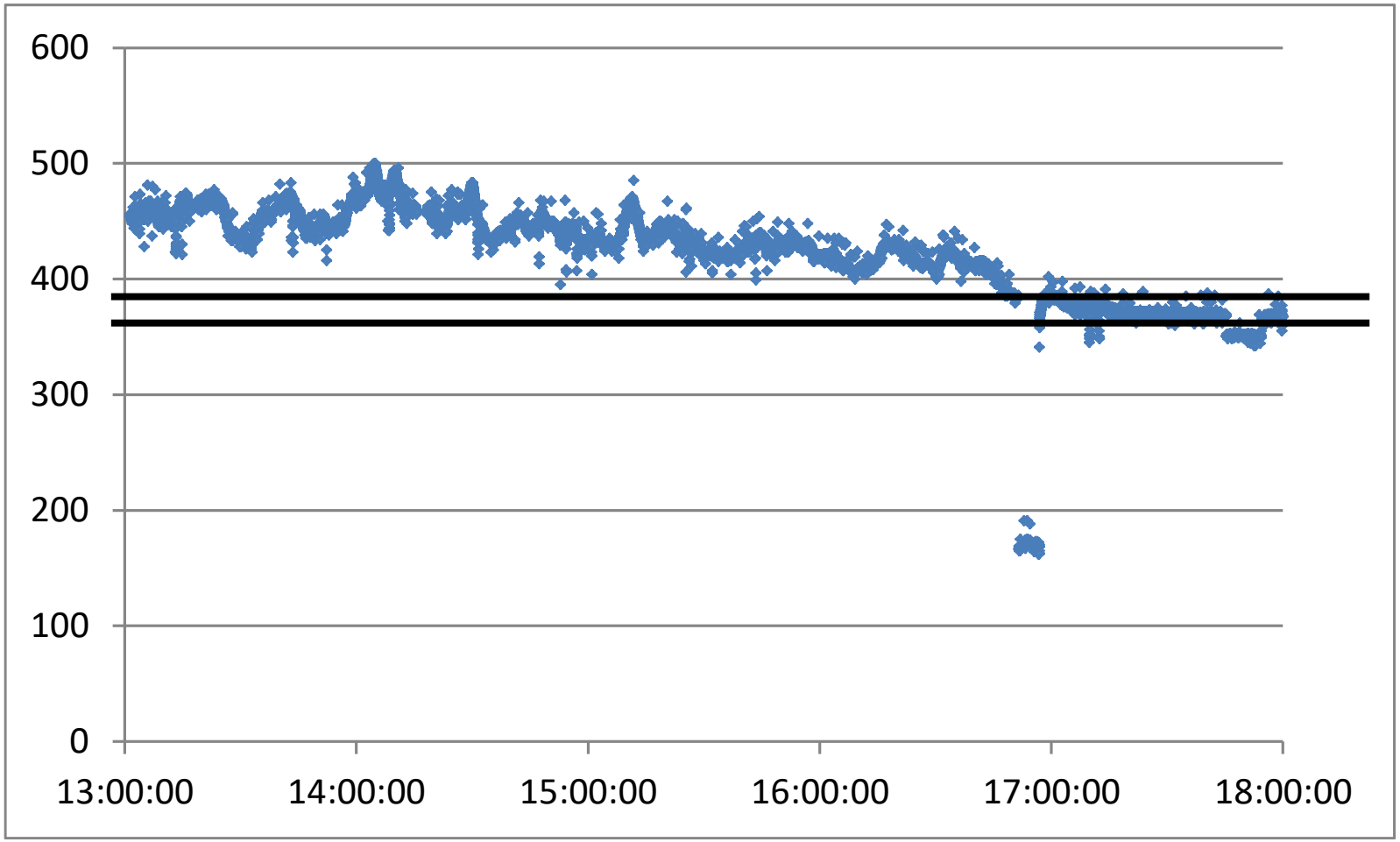
	Algorithm Positive	Algorithm Negative
Real Positive	24 true positives	3 false negatives
Real Negative	2 false positives	50 true negatives



Validation dataset

- The algorithm looks great.
- Statistics supporting it are great.
- But it isn’t.
- Looking at data from other times, the variation in ambient light level means that our threshold is invalid
- We have over-fitted to that one specific data-set during algorithm development.

Look at the threshold



Do you (you probably don't) need to do live in-situ testing

- Many of the systems you are prototyping are designed for use in situations that are not the lab.
  - At the front door of a student house
  - Attached to bridges
  - In coffee shops
  - On boxes containing human organs
  - On wild animals roaming the plains of Africa
- But, you are only building a functional prototype, not a fully deployed device.
- You should be okay prototyping and testing things in the lab, or at least in the vicinity of the lab during lab sessions.

Not in-situ testing things

- You need to create an analogous situation for testing.
  - With evidence that your algorithm could be adapted to run in the real situation
  - Doesn’t worry about things that aren’t relevant to the quality of your algorithm
  - Can be informed by evidence as to what the real world situation is
- In your report, you can discuss:
  - What are the differences between the real-world situation and your testing
  - How you would adapt to the real world situation
- **ANY testing with users who aren't you needs ethics approval**
  - **Talk to Stuart about this – the process is pretty easy.**

Test situation example 1

- **System design:** Detecting knocking on the door of a student house
- **Test situation:** Detect knocking on a table in the lab
- **What might need changing for deployment?:**
  - Material of the door might be different, so knock sensitivity might need to be re-calibrated.
  - Wind and background vibrations from traffic might also require knock detection calibration.
  - Large temperature variations during the day, so any temperature sensitive sensors would need to take this into account.
- **Possible evidence for these changes:**
  - Measurements of the thickness and material of a few student doors?
  - Reference to some kind of science material about how vibrations propagate in different materials?
  - Average day/night temperature readings for a student town, links to datasheets for sensor components showing their temperature response?

Test situation example 2

- **System design:** Sensor in coffee shop, detecting how busy it is
- **Test situation:** A sensor left in the lab
- **What might need changing in deployment?:**
  - Behaviour of people in the lab might be different, for example they may stay longer on average.
  - If prototype uses weekly historical data, weekly rhythms of the coffee shop are likely to be different, new historical data will be needed.
- **Possible evidence for these changes:**
  - Manually recorded data as to average stays in coffee shop vs lab.
  - Brief manually collected sample of activity of the coffee shop versus the activity in the lab at a similar time.

Test situation example 3

- **System design:** Detection of what is happening to human organs in transport
- **Test situation:** Sensors on a small box in the lab, wobbling the box to create 'unsuitable vibrations' that might damage the organ. Using some small object in the box falling over as a measure of what is unsuitable vibration.
- **What might need changing in deployment?:**
  - Need to know how much vibration actually damages the organ – would need to acquire some organs (e.g. animal hearts), create vibrations, and measure damage by inspecting tissue. This would allow re-calibration of the sensing algorithm.
- **Possible evidence for these changes:**
  - Is there any research literature as to organ damage during transportation that you could cite? (yes, there was)

Test situation example 4

- **System design:** Detection of what is happening to animals roaming the plains, including their heart rate and movement, and estimation of whether they are dead (or whether a sensor has failed).
- **Test situation:** Heart rate sensor, and some kind of motion sensor, attached to the developer. At points, sensors are unplugged whilst the developer continues moving to simulate sensor failure, or slowly removed and the develop stops, to simulate death.
- **What might need changing in deployment?:**
  - Need to know what actually happens to the heartbeat signal when an animal dies (and whether we can tell the difference between a violent death, old-age / illness death and sensor failure.) Can't simulate these reliably for the test situation.
- **Possible evidence for these changes:**
  - Literature on animal physiology for the types of animal being considered.
  - Literature on the behaviour of pulse sensors, and the expected pulse trace in various situations.

Summary

- Break down your testing
  - Sensor characteristics, filtering etc.
  - Algorithm logic
  - Whole system testing
- Present this using statistics & graphs.
  - To demonstrate how well your system works
  - So that it would be possible to compare with another similar system.
- Think about how to make test situations which are mostly analogous to your deployment situation.
- Consider what would need changing for real deployment (and provide evidence for this if possible)