Sensor fusion with subjective logic

van Wijk, R.G.H.

Award date:
2018

Sensor Fusion with Subjective Logic

Master Thesis

Rob van Wijk

Supervisor: Tanır Özçelebi
Tutor: Aaqib Saeed

Committee: Tanır Özçelebi
Majid Nabi Najafabadi
Mike Holenderski

November 6, 2018

Summary

Electronic sensors are becoming ever smaller, cheaper and more ubiquitous. Thanks to wireless networking, they can be installed everywhere in our everyday environment. Turning all those sensor values into conclusions to act upon can be a complex task. The situation becomes even more challenging when taking into account that sensors are imperfect devices; the data they return can be inaccurate, wrong or even missing. One solution is to use multiple sensors and fuse their values. This gives more data to work with, but it also makes the problem even more complex. In this thesis we investigate whether an approach called subjective logic is a suitable tool to help us solve this problem. We will use two different scenarios as examples. Our goal in both cases is the same: through the use of multiple sensors at once, obtain more accurate results than can be provided by any single sensor on its own.

The reason we expect subjective logic to be able to help is its use of composite values, which store detailed information about the phenomenon being observed. It keeps track of the amount of evidence for each possible conclusion, thus letting us easily combine sensor data from multiple sources. Since it also contains information about how much uncertainty there is, we are able to merge data from sensors in such a way that the one which produces the most accurate results has the largest influence on the fused value. In order to determine whether subjective logic is indeed a good tool for sensor fusion, we will investigate two scenarios. The difference between the two scenarios is the kinds of sensors used.

In the first case we will use a variety of sensors which measure different phenomena. All of these phenomena provide evidence for or against the condition "a person is in the room". When we only use a single sensor to base our prediction on, it is possible to get incorrect results if the relation between the phenomenon being measured and the condition we are interested in does not act in accordance with our model. For a complex phenomenon this might very well happen from time to time. Through the use of multiple different sensors, we hope it will be possible to make an accurate prediction, even when one sensor returns unclear or misleading values. When we fuse data from multiple sensors, we expect to almost always receive a majority of clear data, thus letting us make the correct prediction in more cases than a single sensor can.

The second test investigates locating a person within a room. For this scenario we will use two identical sensors. Each of these on its own can measure where a person is, but the accuracy of this sensor is limited. We will investigate whether fusing the data from both of them will result in a more accurate location. Apart from just combining the current data, the use of subjective logic also makes it easy to take previous measurements into account, thus giving us an additional method of improving the accuracy of the estimated location. These sensors contain a built-in algorithm for determining the location of a person, which gives us the ability to compare its accuracy against the accuracy of our own approach.

Acknowledgement

This thesis was written to finish my degree at Eindhoven University of Technology. I would like to thank my supervisor Tanır Özçelebi and my advisor Aaqib Saeed for all their help during this project. I also want to thank Xetal for providing the sensors which were used in the second half of this research.

SCOTT (www.scott-project.eu) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway.

Contents

1 Introduction

1.1 Background

Electronic devices keep getting smaller and cheaper. Together with improvements in wireless networking, it has now become feasible to install a multitude of electronic sensors in houses and offices, where they continuously monitor their environment. These "smart buildings" offer a range of possibilities. Common goals include better safety, the reduction of heating and cooling costs and improvements to their users' comfort. With many sensors providing input, it is not immediately clear how to draw conclusions from the data gathered. This is especially true if we take into consideration sensors which are not fully reliable and may sometimes even return incorrect values.

The standard workhorse of computer science, boolean logic, does not lend itself very well to this application. An interesting alternative is so-called *subjective logic*. Whereas boolean logic is restricted to just *True* and *False* ("ones and zeros"), subjective logic allows for the use of the real ("floating point") numbers. Furthermore, it does not only record what is known, it also explicitly keeps track of how much uncertainty there is. This makes it a good fit for processing data which is not perfectly reliable. The increased expressiveness is especially helpful when multiple sensors are used together. In that case, subjective logic provides a much richer way to combine multiple values into a single conclusion.

1.2 Subjective logic

In the previous section we described some of the useful properties of subjective logic. This section will provide a short description of the way in which subjective logic actually works and how it achieves those properties. Note that only parts relevant to this work will be covered. Readers interested in a thorough understanding of all aspects of subjective logic are referred to [7].

At its core, subjective logic is about the question "how much do we trust a certain statement to have a certain answer?". These questions, called *trust statements*, consist of a statement and its possible answers. A simple example, which we will come back to later, could be used by a thermostat: statement "the heater should be active", with possible answers "true" and "false". If we have a more advanced heater, we might instead use the trust statement "the heater should be in state ...", with possible answers "inactive", "active at low temperature" and "active at high temperature". Statements with even more possible answers are allowed as well. We do place a few restrictions on the set of answers though. The most important constraint is that the set of answers should be non-overlapping and complete. In other words, at all times, exactly one answer has to be true; it is not possible for two answers to be true at the same time, nor is it possible for no answer to be true. As a consequence, we require each trust statement to have at least two answers; if only a single answer were possible, we would not need to do any calculation and could immediately conclude that answer has to be true. Furthermore, we do not allow the set of answers to contain an infinite number of elements. Finally, we require the set of answers to be fixed; it is not possible to dynamically add or remove elements.

The next question is how we determine the amount of trust we place in each possible answer. For this goal subjective logic uses *evidence*. For each trust statement, we have two or more variables (one for each possible answer, to keep track of the amount of supporting evidence for it) and a constant (related to uncertainty; this will be explained in detail later). In the simplest case, we have three values: r (a variable representing evidence for the first answer), s (a variable representing evidence for the second answer) and W (the constant). For some trust statements, the answer will be constant. For instance, at the end of a production process, we are interested in the trust statement "the product has the correct weight", with answers "true" and "false". If our scale is slightly inaccurate, we could weigh the product a few times to gain confidence.

Even though not all measurements might agree, the correct answer will not change. In this case, all measurements have the same relevance and we can simply count the measurements. We increment r for each measurement which suggests the statement is true and increment s when a measurement indicates the statement is false. However, in many cases (including the cases considered in the body of this work) we are interested in statements for which the correct answer will change over time, such as the example of the thermostat. In such cases, processing the evidence values requires a more complex approach.

Let us consider the thermostat example in more detail. It consists of a sensor, some logic and a heater (but no device for active cooling). The sensor records the current temperature and the trust statement we are interested in is "the heater should be active", with possible answers "true" and "false". Let us also assume there is a reason not to base the decision solely on the latest measurement (we might be using a cheap sensor which provides intermittent spurious results for instance). The values most recently recorded by the thermometer are the most relevant; slightly older values quickly become less relevant and very old values are completely irrelevant. In other words, as evidence ages, its relevance decreases. A simple solution would be to record the n most recent values, but a more elegant solution exists. If we increase r or s as normal after each measurement, but also decrease both of them over time, we get the desired result, while only having to store two values. If we multiply both values by a number between zero and one every time a new measurement is recorded, the contribution of older measurements decreases, decaying exponentially, without the need to store all of them. The details of this method are described in [2].

Above we have described how to collect evidence values. In order to fully utilise subjective logic, we need a more standardised representation. For this we turn evidence into an *opinion*. Like evidence, an opinion consists of two or more values relevant to what we are measuring and an extra value for the amount of uncertainty. Each of those values is assigned a probability, in the $[0, 1]$ range, such that the total sums to 1. The simplest opinion, matching the simplest evidence above, consists of three values: b (representing belief in the first answer), d (belief in the second answer) and u (uncertainty). The transformation from evidence to opinion is straightforward, each part of the opinion is the fraction of its corresponding evidence relative to the sum of all evidence together with W:

$$b = r/(r + s + W)$$

$$d = s/(r + s + W)$$

$$u = W/(r + s + W)$$

Here we see the use for this extra constant: thanks to W, opinions based on little evidence are different from opinions based on more evidence, even when the ratio of r to s is the same. Technically speaking, W represents *non-informative prior weight*. A detailed explanation and derivation can be found in section 3.5.2 "The Dirichlet Multinomial Model" of [7]. Here we will only mention the eventual conclusion: $W = 2$, which gives us:

Example	Evidence			Opinion		
	r	s	W	b	d	u
Little evidence	1	1	2	0.25	0.25	0.50
More evidence	9	9	2	0.45	0.45	0.10

As the mathematically inclined might have already noticed, it also prevents division by zero when "empty" evidence ($r = 0$, $s = 0$) is turned into an opinion.

Now that we have an opinion, we need to be able to draw a conclusion from it. We can use a coin flip as a simple example, with "the coin comes up heads" as the condition we are interested in. We flip a coin a few times and tally the heads and tails. After a few flips we might have $r = 3$ and $s = 5$. Now our *belief* in "the coin comes up heads" is equal to

$r/(r+s+W) = 3/(3+5+2) = 0.30$. But that is not our *expectation*. In the expected outcome there is no place for uncertainty. Therefore, we have to assign the amount of uncertainty to belief, disbelief or a combination of both. Deciding how to divide the uncertainty is where so called *base rates* come in, denoted by a. Each value in the domain has its own base rate, such that the sum of all base rates is equal to 1. For a coin flip, if we have no evidence (no information to the contrary), the best we can do is assume the result will be fifty-fifty, so both belief and disbelief get the same base rate: $a_r = a_s = 0.50$. In the example above, that means our expectation the coin will come up heads is $(r + W * a_r)/(r + s + W) = (3 + 2 * 0.50)/(3 + 5 + 2) = 0.40$. Alternatively, we can do the calculation using belief and uncertainty, so long as we take the correct fraction of uncertainty: $b + u * a_b = 0.30 + 0.20 * 0.50 = 0.40$. Both approaches will result in the same value, which is called the *projected probability*.

So far we have talked about one opinion at a time. Within subjective logic, many operations exist which work on multiple opinions. Even though many more operators exist, in this work we will only be using fusion operators. When the domains of two opinions are identical, a fusion operator can combine them into a single opinion. In [7] the formulas are provided with which to perform this operation on opinions, but in its simplest form (called the *cumulative fusion* operator) the effect is easier to explain by looking at it in evidence notation. From this point of view, cumulative fusion simply sums the evidence parameters. In other words, the result of this fusion operator is identical to calculating $r_1 + r_2 = r_{\text{fused}}$ and $s_1 + s_2 = s_{\text{fused}}$, then turning the fused evidence into an opinion.

1.3 Problem description

We want to investigate whether subjective logic is a good tool to use for sensor fusion. In order to determine this, we will look at two scenarios, each representing a different use case for sensor fusion. In both cases, we are interested in groups of sensors which all provide evidence for or against the truth of a single condition. This restriction is necessary, because without it fusing makes no sense; fusing a sensor which measures the temperature in the living room and a sensor which detects whether there is any coffee left at the office gives a result which has no meaning. The difference between the two scenarios is the kinds of sensors used to monitor that single condition. In the first case, we will use dissimilar sensors; the different sensors measure different quantities, different phenomena. All of these different phenomena, directly or indirectly, provide evidence regarding the truth of the condition under consideration. This case will be referred to as "occupancy detection", since that is the example we will be using. The second case uses sensors which all measure the same phenomenon, potentially in different ways. In the case of two temperature sensors, they could simply measure different locations on the same object. When using satellite navigation, one sensor could use GPS and the other Galileo. In our case however, we will use two identical sensors, which observe the same area. The difference is that they perform their measurements from different directions and different locations. For this case we use "location detection" as our example.

1.3.1 Occupancy detection

Here we consider a group of dissimilar sensors which observe a single condition. We want to know whether multiple sensors working together can produce a result which is more accurate than that produced by either sensor on its own. The idea is that when one sensor cannot get a reliable answer (it has high uncertainty), another sensor can help to still get the correct answer. If it also happens that the other sensor produces unreliable answers when the first sensor does have the correct answer, both of them in isolation will sometimes produce incorrect answers, but through cooperation they might still be able to find the correct answer most of the time. By automatically keeping track of which sensors have high confidence in their result and which are uncertain, subjective logic is an elegant approach to this problem.

1.3.2 Location detection

In this scenario multiple sensors of the same type are used. When such sensors always return perfect results, this provides no benefit, but we are interested in imperfect sensors. Furthermore, we want to investigate more complex sensors and more complex conclusions. Instead of looking at sensors which simply return a single scalar value, which we turn into a single yes-or-no answer, we want to investigate sensors which return a "richer" set of values, in this case two-dimensional location data (described in detail in section 1.4.2). This should increase our ability to use subjective logic to a much greater extent and let it perform a more elaborate analysis.

1.4 Methodology

1.4.1 Occupancy detection

In order to investigate this test case we will use a publicly available data set, collected by [3]. This data was created by querying all sensors four times a minute, then storing the average value of each variable once a minute. The recorded variables are temperature, absolute humidity, relative humidity, amount of CO2 and light intensity. Apart from the sensor readings, an additional column of data was entered by the researchers manually, denoting whether or not at least one person was present in the office at the given time. Hence the name "presence detection"; no attempt is made to determine where the person is in the room, or even how many people are present, only whether or not the office is occupied at all. This data set is complete in the sense that for each record all values are known.

We will construct a model for each separate sensor, which uses the output from that sensor to estimate, in the form of a subjective logic opinion, whether or not the room is occupied. Then we will fuse two or more of those opinions together, using the cumulative fusion operator. The accuracy of those fused opinions can be compared against the accuracy of the original opinions, to determine which combination of sensors gives the most accurate results.

One thing to keep in mind is that our goal is to investigate whether sensor fusion can reach better accuracy than any single sensor. That means we specificially do *not* want to develop (near) perfect models for individual sensors; in that case accuracy cannot be improved any further, making it impossible to test our theory. For that reason our goal is to develop models which are reasonably accurate, but which still produce incorrect predictions some of the time.

1.4.2 Location detection

For this test case we will use Xetal Kinsei [1] location sensors. By itself, each sensor can scan a room and determine the location of a person. However, both the precision and the accuracy of that location is dependent on where in the room the person is standing. These sensors will only report locations on a (non-uniform, non-rectilinear) grid, which gets sparser as it gets farther away from the sensor, thus giving a less precise location. Note there is no guarantee that the sensors will report the point on the grid which is closest to a person's actual location. The error in measurement (which increases with distance to the sensor) may be considerably larger than the distances between points on the grid. Therefore the reported locations also become less accurate when moving away from the sensor. While these sensors have high reliability, there are still intermittent cases of them failing to detect a person for a short while. Due to these reasons, multiple sensors working together should allow for more accurate results.

Note these sensors can return two kinds of output relevant to our research. The one we are most interested in gives a list of locations where a person appears to be, together with a confidence score for each of those locations. This list is subject to false positives, false negatives and especially to detecting a single person multiple times, in locations close to one another. This is an instantaneous measurement; it does not take into consideration the previous measurement at all and simply tells us what is detected right now. The other output available, which seems

to be intended for regular use, takes care of a lot of preprocessing and then returns a list of locations where a person is detected. This preprocessing filters out intermittent false positives, most false negatives as well as combine multiple, closely grouped points into a single detection. Unfortunately it is not possible to describe the way this algorithm works, since it is proprietary. We have had no access to it; the approach given in this paper to perform the same task consists entirely of own work. The first output will be used as input to our own algorithm. The second output is used for comparison, to test whether our results are more accurate than the built-in solution.

1.5 Research questions

RQ 1 Given multiple, dissimilar sensors, all (indirectly) observing the same phenomenon, can we use subjective logic to develop a sensor fusion algorithm which is more accurate than the single best sensor?

RQ 2 Given one Xetal Kinsei sensor, can we use subjective logic to develop an algorithm which uses the same raw input as the proprietary algorithm, but which gives a more accurate location as output?

RQ 3a Given multiple, similar sensors, can we use subjective logic to develop a sensor fusion algorithm which is more accurate than the single best sensor?

RQ 3b Given multiple, similar sensors, can we use subjective logic to develop a sensor fusion algorithm which is more accurate than the (weighted) average of the single sensors?

2 Related work

The related work is split into three parts. First we have the use of subjective logic for sensor fusion in general. Next we look at other work using sensor fusion for occupancy detection and finally the use of sensor fusion for location detection.

2.1 Subjective logic

While [7] proposes a number of fusion operators, these are only defined as binary operators. Since some of them are non-assosiative, this is a big limitation. This problem is solved in [13] by generalising the definitions and providing n-ary versions of these operators.

It seems not much work has been done on this specific topic yet. While [12] *describes* the principle of using subjective logic to perform sensor fusion, actually implementing a system and testing its performance is left as future work. An approach which at least includes subjective logic is discussed in [11]. However, that work only uses subjective logic *opinions* to provide the conditional probabilities needed by the Bayesian network which forms the core of their approach, whereas for us subjective logic will be the primary tool. Another important difference is that their work considers what we would call a trust statement with a constant answer. As explained in section 1.2, here we will only investigate trust statements which have a variable answer. Thus we can collect evidence over time and retain it from one measurement to the next, instead of having to rely solely on a single evaluation.

In [14] an approach which is very similar to subjective logic, *Dempster-Shafer Belief Theory*, is used to perform sensor fusion. While they do get a good result, that paper only shows a single example, so it is unclear whether their approach will always manage to turn the use of additional sensors into improved accuracy.

2.2 Occupancy detection

Occupancy detection is a popular area of research. This is caused by its potential to significantly lower the operating costs of climate control in office buildings. Light, CO2 and motion sensors are popular choices, but various other sensor types are used as well; a description of many possibilities is included in [8]. Many authors explicitly mention we should not rely on just one type of sensor and a multitude of approaches have been suggested to combine multiple sensors. In [5] a single sensor manages to reach an accuracy of 97.9% (for their data set), which can be improved to 98.4% with sensor fusion. Note they also find that adding more sensors is not always beneficial; it can potentially *decrease* accuracy instead. This is a possibility we will also have to be aware of. An extensive analysis is performed in [17], which investigates various kinds of sensors as well as multiple ways of fusing them. They too note that using more sensors is not necessarily better. Another review of various approaches was done by [16].

2.3 Location detection

An enormous amount of work has been done on this topic. However, much of it performs a subtly different task from what we envision. One line of research works on determining *which* room a person is in, whereas we want to know *where* in the room the person is located. Other research works with sensors which, by themselves, cannot actually determine the location of a person, only their bearing relative to the sensor. In these cases, multiple sensors do work together in order to provide a location (using triangulation or similar techniques), but we want to fuse sensors which are capable of determining a person's location on their own. To illustrate: using multiple GPS satellites to find the location of a single GPS receiver is not what we are after. Readers interested in either of these areas could start with [6].

A seemingly very similar question would be to use multiple receivers for independent satellite navigation systems. Fusing the locations calculated by each receiver into a single, more accurate

location would be exactly the kind of scenario we are interested in. Much research has been done on exactly this topic, such as in [10]. These navigation systems not only provide location data, they also provide an accurate time signal. This enables their use for synchronising time between multiple systems, as shown by [18]. Note that work relies on specific details of the internal workings of these systems, whereas in our work we do not use detailed knowledge of the underlying mechanisms by which our sensors operate, only the data they report.

Going back to the indoors situation, one approach is provided by [9]. Unfortunately, they only explain their approach, but do not give performance figures. While [15] is not exactly doing location detection in the way we are talking about, they do provide a nice example of sensor fusion.

3 Experimental setup

This section explains in detail how we obtained the data for the experiments described in section 4. It also includes details of the content of the data sets.

3.1 Occupancy detection

For this case, a data set was already available, which was used as-is [3]. This section will describe the structure of this data.

3.1.1 Data format

The data has been stored as a CSV file. The first column contains an id number for each data sample. The second column has a timestamp. Both of these columns are not used by our code. The second column was however used to verify the data samples were indeed stored in-order. The remaining columns contain five sensors and the ground truth. The data set consists of three files. File *datatraining.txt* consists of 8143 lines of samples and has been used as training data. No file named "validation" is present, so we use *datatest.txt* (the smallest file, with 2665 samples) for that purpose. Finally the *datatest.txt* file with 9752 lines is used as actual test data. Note that all data (in all three files) is complete; every single sample contains data in every field. The five sensors and the range of their values (in the training data set) is as follows.

Table 1: Ranges of input in training data

Name	Lowest value	Highest value
CO2	412.75	2028.50
Humidity	16.7450	39.1175
Humidity ratio	0.002 674	0.006 476
Light	0.000	1546.333
Temperature	19.00	23.18

To get a feel for how the data behaves, we also plotted the data set. Due to the differences in ranges, fitting all functions into a single graph required scaling each of them to fit in the $[0, 1]$ range. The ground truth is on the secondary axis and only takes on the values 0 and 1. The entire training data set is shown in figure 1. A closeup of a part of it can be seen in figure 2.

Figure 1: Visualtion of full training data set

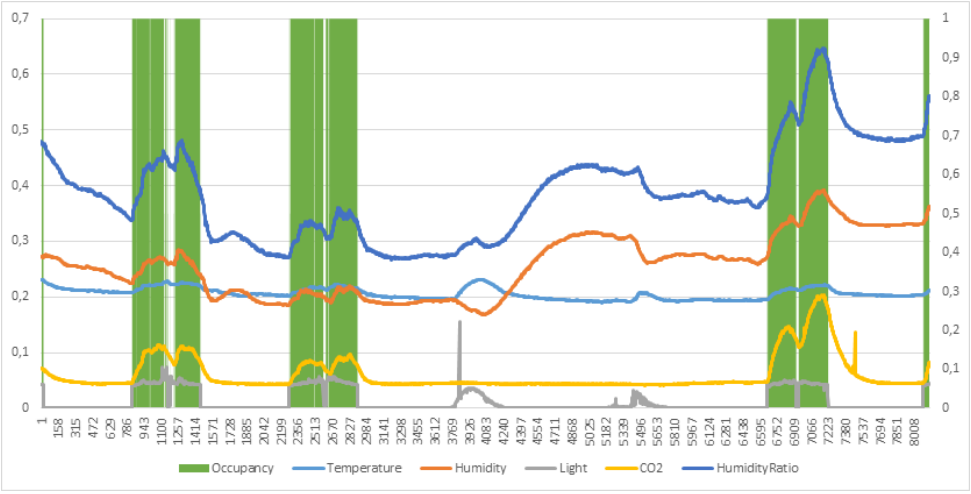
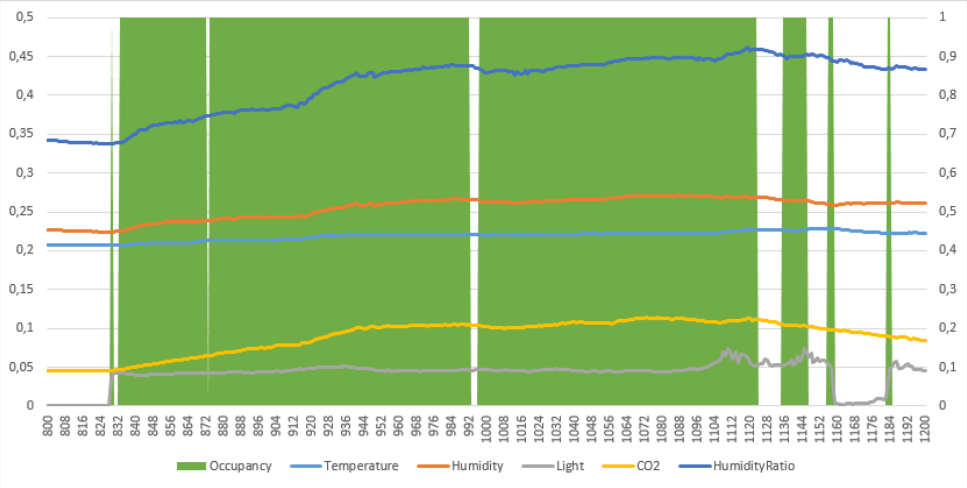


Figure 2: Visualtion of part of training data set



3.1.2 Format error

Before we begin, the data file needs one tiny correction. The first line of the file contains the column headers. For some reason the first column is missing its header, causing all other headers to be misaligned. This can be corrected by adding a header for the first column. It does not matter what it says, even an empty string suffices; it just needs to be there.

3.1.3 Performance metric

We will compare performance based on the F_1 metric. This section will explain how this metric is calculated and why it was chosen as the metric to use in this case.

For each sample we know the ground truth, the correct answer. That means we can compare it to the prediction given by a model. Using "False" to denote an empty room and "True" for an occupied room, there are four possibilities.

Ground truth	Model	Name
False	False	True negative
False	True	False positive
True	False	False negative
True	True	True positive

Next we introduce the concepts "precision" and "recall":

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

It might be easier to get an intuition for what is going on by thinking of *precision* as "which fraction of the predicted positives are true positives?", or alternatively "which fraction of the predicted positives have been classified correctly?". And *recall* is "which fraction of the actual positives are true positives?", or "which fraction of the actual positives have been classified correctly?". Due to this definition, the possible range of both *precision* and *recall* is $[0, 1]$ [1], where higher values are better. Scoring very high on either *precision* or *recall* is easy. Being

[1] This assumes at least one true positive and one actual positive exist in the data set; otherwise we end up with a division by zero.

very selective about returning *True*, only reporting the samples you have very high confidence in, will give high *precision* (at the cost of low *recall*). Likewise, only reporting *False* when you are sure about it will give high *recall* (at the cost of low *precision*).

In many cases, including the one we are looking at here, it is preferable to have both good *precision* and good *recall*. Lowering one a little bit if that allows a big improvement in the other is considered advantageous. Maximising both *precision* and *recall*, finding a balance between them and getting a single number as a performance score with which to compare multiple solutions is the goal of the "F_1 score". This is calculated as:

$$F_1 = 2\,\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

This too has a range of $[0, 1]$ where higher values are better. When showing the performance of each model we will only report the F_1 score and omit the values for precision and recall.

3.2 Location detection

In this case, no existing data set appears to be available, therefore we had to collect our own data. This section will explain the steps taken.

3.2.1 Data collection

An empty space was created in our office, for use as a test area. Two identical sensors, of the type described in section 1.4.2, were used for measurement, one adjacent to a wall, the other parallel to the windows. In the test area a number of markers were placed on the floor, labelled "A" through "H" [2]. The outputs of both sensors were recorded approximately four times a second, together with a timestamp. Ground truth was recorded using a smartphone carried by the test person. A simple script randomly selects a marker to walk to; once there it logs the timestamp and the name of that marker, then selects the next one. This results in one entry every few seconds.

Each sensor uses its own location as the basis for a coordinate system. Since the sensors are in different locations, their coordinate systems do not match. Prior to data collection, the location of each sensor was determined, using the same coordinate system as for the locations of the markers. These locations were used by the data collection script to translate all coordinates from sensor coordinates to our own coordinates, prior to being stored.

Due to limitations of the sensors used, each sensor had to be connected to its own laptop. Because of this, it was not possible to collect data samples from both sensors in sync. Thus the timestamps in one file do not match the timestamps in the other file. Ground truth is recorded whenever the test person reaches the marker, not in fixed time intervals at all. This makes analysing the data more complex. A detailed explanation is given in section 4.2.3.

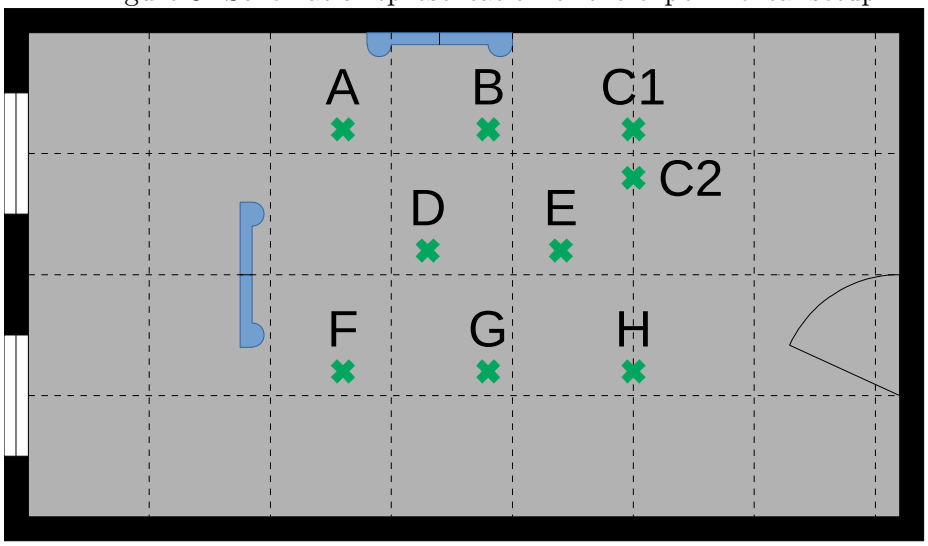
3.2.2 Data format

Two types of files were used to store the data about each experiment. One file contains ground truth information. The structure of this file is very simple: each line contains the information about a single data point: date, time and the name of the marker (which can be converted to a location using a lookup table). The lines appear in order of increasing timestamps. No other information is included in these lines and no other lines appear in the file. The other file type contains the recorded sensor data, with one file for each sensor.

This second file type starts with a few lines containing general information (including which sensor recorded it). This can be skipped over by our analysis code. Then the actual data is

[2] After the first day of data collection, it was noted marker "C" was just slightly too far away from the window sensor, often causing the test person not to be detected in this location. On the second day "C" was moved slightly to correct for this. Hence the existence of "C1" and "C2".

Figure 3: Schematic representation of the experimental setup



A	2600	3200
B	3800	3200
C1	5000	3200
C2	5000	2800
D	3200	2200
E	4400	2200
F	2600	1200
G	3800	1200
H	5000	1200

Sensor Wall
3400 4000

Sensor Window
1750 2000

listed, in chronological order. Again we have one line for each measurement. This file only records the time of each sample, the date is omitted (but, if error checking is desired, the date is mentioned in the header). After that we have four, tab-separated, data fields. The first value is the number of people detected, as an integer. The second value is again the number of people detected, but this time as a floating point number (using fractional people to denote uncertainty about the number of detections) [3]. Then we get the detection locations. The third item is the output of the built-in proprietary algorithm. In our configuration it returns an array of four values. Each value in the array is either an empty array (meaning no person detected) or an array of two integers (representing the location where a person was detected). In rare cases, the parent array can contain two non-empty values. Finally we have the raw sensor output. This is an array with a variable number of entries, one for each detection (no empty entries appear). Each detection contains two integers (a location) and one float (a confidence score). When multiple detection are reported, they do not seem to appear in any particular order. While the documentation does not make this promise, it appears as though we can rely on the reported coordinates all being distinct. The exact meaning of the confidence scores is not explained in documentation; all we have been able to determine is that higher values correspond to a higher confidence. No information is available about the possible range either. Looking through the collected data, we find countless occurences of the value 0.9, yet not a single value lower than that. It seems plausible for this to be the lower bound of the range. Values above 3.0 are rare,

[3] At the time data was collected, it was expected we would probably not use either of these values. Since the data was easy to query and store, we collected it anyway, just in case it might turn out to be useful. In the end we indeed did not use it.

with just a handful above 4.0. While the highest we have seen is 4.1, we have no way to be sure 4.2 is impossible. However, even without an exact upper bound, it does seem very likely an upper bound exists and it is close to this value. Finally, the file contains one line at the bottom to mark the end of the file. This line is used to ensure data collection was successful; its absence would denote communication with the sensor failed halfway through the experiment and data is incomplete. All such files have been omitted from analysis.

3.2.3 Data sets

Multiple data sets have been collected. The first few were twenty minutes long, later sets are fifteen minutes each. The following table lists the data sets which were collected successfully. The missing numbers are caused by failed attempts (usually due to the laptop losing its connection to the sensor). The numbers in the *filename* column refer to the filenames used to store each data set. The letters in the *usage* column show the names by which we will refer to these data sets in this report. The alphabetical order of the names is a rough approximation of the increasing difficulty to determine accurate results.

Filename	Usage	Description	Note
#3	Training "C"	*All three:* Intended as good data, but	
#4	Test "C"	marker "C" found to be poorly placed;	
#5	Spare	contains a small occluded region	
#6	Training "B"	Good data	
#8	Test "B"	Good data	
#9	Test "D"	One occluded region	
#11	Training "D"	One occluded region	Ground truth imperfect
#12	Training "A"	Good data; only long stretches	
#13	Test "A"	Good data; only long stretches	
#14	Spare	Good data; only long stretches	

The label "good data" means an attempt to get the best possible data; both sensors have an unobstructed view of the entire testing area. For #9 and #11 one of the sensors deliberately had some items placed in front of it, to occlude a section of the testing area from its view. In all tests up to and including #11, the random selection of the next point to walk to can select any point except the current one. This leads to many shorts walks. From #12 onwards the random selection was changed to choose non-neighbouring points only. This lets the test person walk longer paths and be in motion, as opposed to momentarily standing still on a marker, a larger fraction of the time.

The note regarding #11 relates to the test person stumbling. Since #10 had even bigger problems and due to time constraints not allowing to do another full run, it was decided to use the data from #11 anyway. Even though the effect is expected to be very small, just to be sure we swapped #11 to be used as a training set, with #9 becoming a test set.

3.2.4 Performance metric

For this case we will use the geometric distance between the calculated location and the ground truth location as the performance metric of each sample. The metrics of all the samples in a data set will be combined into a single metric using root-mean-square. This yields a single number for each test, with a lower value meaning a more accurate result.

4 Evaluation

4.1 Occupancy detection

The first case we are investigating is that of multiple sensors, each measuring a different phenomenon. We will consider the example of a room in which various environmental properties are measured, with the goal of automatically determining whether or not a person is currently in the room. In order to determine how well-suited subjective logic is for sensor fusion, we will compare predictions based on a single sensor with predictions based on data from multiple sensors.

A large part of this section talks about the development of the model used to make our predictions. Whenever we analyse data in order to find the best values for parameters, we do so by analysing the training data. Only in the final part of this section, when we consider the eventual results of our model, do we analyse the real test data.

The tables containing all test results are quite long. In order to improve the readability of this section, only the top five of best results will be included in this section. The complete tables containing all results are available in appendix A. Both the full and top five tables follow the same format: the initial columns indicate which sensors were fused and which were omitted from fusion, the last column shows the corresponding F-score.

4.1.1 Model A

The first step is to turn each sensor value into an estimation of whether a person is present. For this goal we use a simple model, inspired by [4]. All sensors return low values for an empty room, whereas an occupied room causes high values. It turns out the same model works well for all five sensors, given appropriate parameters. The model only looks at the previous sample and at the current one (it has no memory beyond that). When the difference between these two samples is larger than a, model-specific, rate of change we say the value is changing quickly. In this case we use the direction of change to make our prediction: dropping values mean the room is empty, rising values mean the room is occupied. When the value is changing slowly, we look at the value itself for our prediction. When it is below a certain lower bound, the room is predicted to be empty; above an upper bound we say it is occupied. Between these two bounds we perform a linear interpolation and return a fraction, to show how likely we estimate it to be for the room to be occupied.

The estimations found by the models are then given to the trust models. The trust model we use here is the one proposed in [2]. Each sensor has its own, independent trust model, with its own value for γ and δ. We use r to denote belief and s for disbelief. For each observation, the following calculations are performed for each consecutive sample:

$$r_{new} = \delta^{1-observation} * r_{old} + observation$$

$$s_{new} = \gamma^{observation} * s_{old} + 1 - observation$$

The value of *observation* is in the range $[0; 1]$, with 0 meaning we strongly believe the room to be empty and 1 meaning we strongly believe the room to be occupied. A value between these extremes denotes uncertainty; for instance 0.9 signifies a pretty strong belief in the room being occupied, but with a little doubt. Values like this occur when the value reported by the sensor is changing slowly, while between the lower and upper bound for that sensor.

The first addend retains a certain fraction (between the parameter and 1) of the previously collected evidence and the second addend collects new evidence. Here it becomes clear why we do not restrict ourselves to returning 0 or 1, but also use values in between. An example will make this more clear. To keep things simple, we will use $\delta = \gamma = 0.5$ for this example and assume $r_{old} = 10.0$ and $s_{old} = 2.0$. The following table demonstrates how return values between 0 and 1 cause meaningful and useful results.

Table 2: Example of using fractional values for *observation*

observation	r_{new}	s_{new}
0.0	5.00	3.00
0.1	5.46	2.77
0.3	6.46	2.32
0.5	7.57	1.91
0.7	8.22	1.53
0.9	10.23	1.17
1.0	11.00	1.00

With the above model, we have five parameters for each sensor: lower bound, upper bound, rate of change, δ and γ. We determine which values to use for these parameters by exhaustively searching the, quantised, parameter space. For δ and γ we search from 0% to 100%, in 1% increments. For the other parameters we cannot use fixed bounds and step sizes, since the magnitudes of the values reported by each sensor differ greatly. Therefore we first analyse the entire file. For each sensor we keep track of the lowest value reported, the highest value reported, the magnitude of the smallest change between consecutive values and the magnitude of the largest change between consecutive values. We search for a lower bound and upper bound from the lowest to the highest value reported, with the restriction that the upper bound is strictly larger than the lower bound and with a step size such that there are between one hundred and one thousand steps. The search for the best rate of change goes from the smallest change to the largest change, also with a step size which creates between one hundred and one thousand steps. The exact code used can be found in appendix B.2.4.

We use the approach described above to analyse the training data. This gives us the following values for the parameters.

Table 3: Parameters for model A

Name	Lower bound	Upper bound	Rate of change
CO2	410.0	1010.0	3.0
Humidity	38.7	40.7	0.16
Humidity ratio	0.004 270	0.006 070	0.000 001
Light	200.0	600.0	100.0
Temperature	19.2	23.2	0.008

Table 4: Parameters for model A

Name	γ	δ
CO2	0.83	0.90
Humidity	0.32	1.00
Humidity ratio	0.91	0.93
Light	0.18	0.48
Temperature	0.90	0.90

In order to combine the results from different sensors and get a combined prediction, we have to fuse the results from various sensors into a single value. Since [7] defines his fusion operators on opinions, not evidence, we first convert our evidence into opinions. As was shown

in section 1.2, but this time using descriptive names, that means:

$$occupied = r/(r + s + W)$$

$$empty = s/(r + s + W)$$

Now we can fuse the results from the various sensor together into one overall conclusion. For this we use the cumulative fusion operator. Ignoring edge cases which will never happen in our situation, fusing a and b together into c, all of which have domain X, entails the following:

$$\forall x \in X \quad \text{belief}_x^C = \frac{\text{belief}_x^A * \text{uncertainty}^B \ + \ \text{belief}_x^B * \text{uncertainty}^A}{\text{uncertainty}^A \ + \ \text{uncertainty}^B \ - \ \text{uncertainty}^A * \text{uncertainty}^B}$$

$$\text{uncertainty}^C = \frac{\text{uncertainty}^A * \text{uncertainty}^B}{\text{uncertainty}^A \ + \ \text{uncertainty}^B \ - \ \text{uncertainty}^A * \text{uncertainty}^B}$$

Even though it is not immediately obvious, this operator turns out to be associative, so the order in which we fuse values does not matter.

4.1.2 Model A, results

In order to determine which combination of sensors works best, we simply try all $2^5 - 1 = 31$ possible combinations [4]. The program which generates the following results can be found in appendix B.2.1. The full results are available in appendix A.1.1, as tables 26 and 27. Table 5 listed here shows the five best results.

Table 5: Model A, top five results, using all sensors

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.981
Used	–	–	Used	–	0.923
Used	–	Used	Used	–	0.902
–	–	Used	Used	–	0.878
Used	–	–	Used	Used	0.873

To summarise, the light sensor by itself performs best. The rest of the top five is made up of the light sensor fused with various other sensors, which decreases its accuracy by various amounts. In other words, fusing multiple sensors is not only unhelpful, it actually decreases accuracy.

What happens when we say the light sensor is just too reliable and it skews our results? Or, looking at it another way, what would the table have looked like if the light sensor had not been part of the data set? Removing all entries which include this sensor creates that situation. The full list of those results is available as table 28 in the appendix; table 6 shows the five best results.

While the top result is a fusion of three sensors, a look at the F-score shows we are very close to a three-way tie. This top three is in some ways similar to the previous table: one sensor which is always used and other sensors get fused in or not. Whereas in the previous case the accuracy decreased, here it remains almost the same. That is better than fusion causing a drop in accuracy, but what we are hoping for is an actual increase.

If we also remove the temperature sensor from consideration, we get table 29 and table 7. Here we finally see a situation where fusion actually helps. But at this point, we are clearly cherry-picking.

[4] One of the possibilities is combining none of the sensors. While this was calculated by our code (and as a sanity check, manually verified to perform terrible), it has been omitted from tables of results.

Table 6: Model A, top five results, light sensor omitted

CO2	Humidity	Humidity ratio	Temperature	F-score
Used	–	Used	Used	0.742
–	–	–	Used	0.737
–	–	Used	Used	0.737
Used	–	Used	–	0.643
Used	–	–	Used	0.628

Table 7: Model A, top five results, light and temperature sensors omitted

CO2	Humidity	Humidity ratio	F-score
Used	–	Used	0.643
–	–	Used	0.526
Used	–	–	0.455
Used	Used	–	0.403
Used	Used	Used	0.402

4.1.3 Model B

Maybe our model, simple as it was, performed too well and did not leave enough room for further improvement? That would make it impossible for sensor fusion to increase the accuracy of the results. In order to investigate this possibility, we create a new model, which will perform a little more poorly. The code of this model is identical to the previous model; the difference is entirely in the choice of parameters. When creating model A, we used a very small step size, so that we are almost guaranteed to find a result very close to the best possible one. We now make that model less accurate by rounding its parameters. We will use the following values for our new model.

Table 8: Parameters for model B

Name	Lower bound	Upper bound	Rate of change
CO2	400.0	1000.0	3.0
Humidity	39.0	40.0	0.16
Humidity ratio	0.004	0.006	0.000 001
Light	200.0	600.0	100.0
Temperature	19.2	23.2	0.008

Table 9: Parameters for model B

Name	γ	δ
CO2	0.80	0.90
Humidity	0.30	1.00
Humidity ratio	0.90	0.90
Light	0.20	0.50
Temperature	0.90	0.90

4.1.4 Model B, results

The full tables of all results are provided in appendix A.1.2 as tables 30, 31, 32 and 33; the corresponding top fives are shown here in tables 10, 11 and 12.

Table 10: Model B, top five results, using all sensors

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.981
Used	–	–	Used	–	0.924
Used	–	Used	Used	–	0.907
–	–	–	Used	Used	0.874
Used	–	–	Used	Used	0.873

Table 11: Model B, top five results, light sensor omitted

CO2	Humidity	Humidity ratio	Temperature	F-score
–	–	–	Used	0.737
–	–	Used	Used	0.735
Used	–	Used	Used	0.724
Used	–	–	Used	0.612
Used	–	Used	–	0.586

While the exact F-scores differ a bit, the overall results follow the same pattern as the original results. In short: looking at the single best sensor is the way to go; fusing multiple sensors only hurts the accuracy of the results.

Table 12: Model B, top five results, light and temperature sensors omitted

CO2	Humidity	Humidity ratio	F-score
Used	–	Used	0.586
–	–	Used	0.508
Used	–	–	0.450
Used	Used	–	0.402
Used	Used	Used	0.401

4.1.5 Model C

Since the approach in the previous section did not help, perhaps we should go the other way. Maybe a more advanced model, capable of giving more accurate results, is needed to enable sensor fusion to give an increase in the accuracy of predictions. We investigate this option by adding two parameters. So far, when turning a subjective logic opinion into an actual prediction, we have looked at whether the amount of belief in "occupied" is larger or smaller than the belief in "empty". In that approach there are no parameters. We can however use an alternative approach. In order to get the most out of our subjective logic foundation, an approach was chosen which not only uses the belief in either result, but also takes the amount of uncertainty into account.

As described in the introduction to subjective logic (section 1.2), projecting an opinion uses the base rate to let the amount of uncertainty affect the resulting probability. While we would normally check whether the result is larger or smaller than one half, we want to give the model one more opportunity to perform just a little bit better and therefore we also turn that threshold into a parameter.

Unfortunately, the number of parameters is growing too large to exhaustively go through all possible combinations with a small step size. Therefore, we first try all combinations using a larger step size. As a second step we go through all of the parameters one by one and try to optimise them further using small steps. The code which performs this search is listed in B.2.5. Note this approach is *not* guaranteed to find the global optimum: we might get stuck in a local optimum. As it turns out, for the humidity ratio sensor, this is exactly what happens.

Table 13: Accuracy comparison of model A and model model C

Sensor	F-score (old mode)l	F-score (new model)	Ratio
CO2	0.919 201	0.920 671	1.001 599
Humidity	0.710 968	0.736 594	1.036 043
Humidity ratio	0.712 198	0.712 112	0.999 879
Light	0.974 723	0.975 540	1.000 838
Temperature	0.743 934	0.748 868	1.006 632

This is not really a problem, since this model can easily emulate the previous model. If we set both *base rate* and *threshold* to 0.50, the new model will always behave exactly as the old model. Since the old model for *humidity ratio* is better than the new model, we will use the old parameters for this sensor. That leaves us with the following parameters.

Even though the other models got only very slightly better, we do have some hope left this approach may still work. Unlike the previous model, we now have a parameter which only comes into play after sensor fusion, so it is optimised separately. This keeps open the possibility of results which have improved accuracy.

Table 14: Parameters for model C

Sensor	Gamma	Delta	Lower Bound	Upper Bound	Rate of change	Base
CO2	0.90	0.90	410.000 000	1210.000 000	3.000 000	0.90
Humidity	0.40	1.00	38.700 000	40.700 000	0.160 000	0.99
Humidity ratio	0.91	0.93	0.004 270	0.006 070	0.000 001	0.50
Light	0.24	0.20	200.000 000	400.000 000	63.000 000	0.90
Temperature	0.90	0.90	19.600 000	23.200 000	0.008 000	0.85

Table 15: Parameters for model C

CO2	Humidity	Humidity ratio	Light	Temperature	Threshold
–	–	–	–	Used	0.48
–	–	–	Used	–	0.90
–	–	–	Used	Used	0.77
–	–	Used	–	–	0.49
–	–	Used	–	Used	0.42
–	–	Used	Used	–	0.56
–	–	Used	Used	Used	0.74
–	Used	–	–	–	0.39
–	Used	–	–	Used	0.41
–	Used	–	Used	–	0.57
–	Used	–	Used	Used	0.56
–	Used	Used	–	–	0.42
–	Used	Used	–	Used	0.45
–	Used	Used	Used	–	0.56
–	Used	Used	Used	Used	0.54
Used	–	–	–	–	0.30
Used	–	–	–	Used	0.46
Used	–	–	Used	–	0.41
Used	–	–	Used	Used	0.55
Used	–	Used	–	–	0.33
Used	–	Used	–	Used	0.45
Used	–	Used	Used	–	0.36
Used	–	Used	Used	Used	0.51
Used	Used	–	–	–	0.39
Used	Used	–	–	Used	0.42
Used	Used	–	Used	–	0.55
Used	Used	–	Used	Used	0.51
Used	Used	Used	–	–	0.41
Used	Used	Used	–	Used	0.40
Used	Used	Used	Used	–	0.53
Used	Used	Used	Used	Used	0.53

4.1.6 Model C, results

See tables 34, 35, 36 and 37 in appendix A.1.3 for full results; tables 16, 17 and 18 contain the top fives.

Table 16: Model C, top five results, using all sensors

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.982
Used	–	–	Used	–	0.968
–	–	Used	Used	–	0.952
Used	–	Used	Used	–	0.924
–	–	Used	Used	Used	0.922

Table 17: Model C, top five results, light sensor omitted

CO2	Humidity	Humidity ratio	Temperature	F-score
–	–	–	Used	0.745
Used	–	Used	Used	0.744
Used	–	–	Used	0.682
–	–	Used	Used	0.665
–	–	Used	–	0.524

Table 18: Model C, top five results, light and temperature sensors omitted

CO2	Humidity	Humidity ratio	F-score
–	–	Used	0.524
Used	–	Used	0.497
Used	–	–	0.459
–	Used	Used	0.369
Used	Used	Used	0.365

The results of model C are just like those for the other models; irrespective of the group of sensors we consider, all the top spots go to single sensors again. Sometimes one of the fused results comes close, but we still have not found a situation where sensor fusion leads to a result with significantly higher accuracy.

Taking everything together, the only fair conclusion for this data set is that generally speaking sensor fusion using subjective logic does not lead to better results than simply relying on the best sensor.

4.2 Location detection

The other case we investigate is that of multiple sensors, each measuring the same quantity. For this we use a room in which two identical sensors, placed in different positions, measure the location of a person. We will determine how well-suited subjective logic is for sensor fusion in this application by comparing the accuracy of measurements from either single sensor with the accuracy of measurements from fusing both sensors.

4.2.1 Fusing proprietary results

As discussed in section 1.4.2, the sensors can provide either raw data for us to process or do some proprietary processing and simply return a location. Since we want to compare the accuracy of non-subjective logic sensor fusion to the accuracy of our subjective logic sensor fusion, we need to fuse the results from the two sensors. Since the sensors themselves do not support this, we will have to come up with our own solution. One option is to simply take the average value. However, since sensor accuracy decreases further away from the sensor, we expect a more accurate result if we use a weighted average, with weights based on the distance from the sensor to the reported location. To test this, we assign a weight of 1 to the sensor which reports the closest location. The weight of the location given by the other sensor is the ratio of the smaller distance to the larger distance. To investigate how aggresive the weighting should be, we also calculate the accuracy for the weight raised to a few different powers. The results of these approaches are given in section 4.2.6.

4.2.2 Synchronising clocks

For each data set there are three separate log files: one for each of the sensors and one for the ground truth. Due to practical limitations, all these files had to be created on different machines, with clocks which were not perfectly synchronised. A manual comparison, at the time of data collection, showed the clock of the computer collecting data from the wall sensor to be approximately 0.5 seconds ahead of the clock on the smartphone recording ground truth and the window sensor is about 8.5 seconds ahead. In order to determine the time difference between the clocks more accurately, we shift the time stamps relative to each other and search for the shift which results in the smallest RMS error between the locations given by ground truth and the sensors.

Out of the data sets intended for training, the ones which have the "cleanest" data are training A and training B, because they do not contain any occluded regions. Therefore they will be used for this purpose. First we try whole second offsets, then we do a more fine-grained search around the optimal value. The conclusion is that we will use an offset of 0.8 seconds for the wall sensor and 8.0 seconds for the window sensor. These values match quite well with the manual observations. The code which produced these results can be found in appendix B.3.4.

Table 19: Wall sensor: RMS error

Time offset	Training A	Training B
-10.00	1373.27	1532.40
-9.00	1473.19	1586.01
-8.00	1595.32	1628.00
-7.00	1687.54	1638.56
-6.00	1704.82	1596.62
-5.00	1620.75	1485.35
-4.00	1425.42	1292.51
-3.00	1124.10	1014.61
-2.00	762.28	689.90
-1.00	497.24	467.93
0.00	609.36	588.42
-2.00	762.28	689.90
-1.90	726.51	663.24
-1.80	693.25	629.62
-1.70	659.05	602.02
-1.60	630.35	574.31
-1.50	599.55	549.20
-1.40	572.49	529.65
-1.30	548.01	506.79
-1.20	527.42	491.58
-1.10	511.63	476.50
-1.00	497.24	467.93
-0.90	489.88	464.29
-0.80	483.33	461.35
-0.70	487.15	466.11
-0.60	492.45	470.83
-0.50	501.55	483.85
-0.40	518.90	498.70
-0.30	532.47	516.62
-0.20	559.48	539.30
-0.10	582.52	560.29
0.00	609.36	588.42

Table 20: Window sensor: RMS error

Time offset	Training A	Training B
-10.00	1053.98	991.09
-9.00	682.44	659.36
-8.00	469.26	452.93
-7.00	676.22	607.14
-6.00	1045.81	929.56
-5.00	1377.62	1229.50
-4.00	1608.60	1450.28
-3.00	1721.98	1584.83
-2.00	1727.80	1645.08
-1.00	1651.08	1649.01
0.00	1532.50	1615.05
-9.00	682.44	659.36
-8.90	650.55	628.10
-8.80	615.64	596.53
-8.70	585.94	571.66
-8.60	557.51	543.85
-8.50	532.03	520.92
-8.40	512.28	500.64
-8.30	492.19	480.75
-8.20	480.43	470.73
-8.10	470.92	458.39
-8.00	469.26	452.93
-7.90	472.14	453.56
-7.80	477.41	453.84
-7.70	491.02	465.31
-7.60	505.52	473.89
-7.50	529.01	488.73
-7.40	552.59	510.15
-7.30	579.03	528.83
-7.20	609.88	556.38
-7.10	639.32	579.90
-7.00	676.22	607.14

4.2.3 Developing a model

Our next step is to design an algorithm, based on subjective logic, which turns the raw data into a location. In reality this is a continuous quantity. However, in order to apply subjective logic, we will quantise the location of a person into a number of regions. Each region will represent the trust statement "the person is somewhere in this region". For these trust statements we can only gather positive evidence, we have no way to explicitly gather negative evidence. In order to fully focus on the localisation aspect, we will a priori assume that a person is definitely present somewhere in the room. This means the uncertainty component of our trust value will always be zero. The trust value will be turned into a conclusion by selecting the region which has the highest amount of positive evidence. When calculating the distance from the ground truth value, we will use the centre point of the region.

Since we will be using distance as the performance metric, it is important we always return a location; after all, the distance between a point and "no data" is not defined. This goal can be reached by using a similar approach as in the *occupancy detection* case; we collect evidence over time but also let old evidence decay. When a sensor temporarily does not detect the person, we

cannot add new evidence, but we can use the retained, older evidence to still report a location.

The next step is to determine how to assign positive evidence to each domain element. For this we use statistics. The true location of the person is taken to be a probability distribution, with the location reported by the sensor as its mean and a parameter for its standard deviation. We assume the true location is normally distributed along both the x and y axes and with a correlation between x and y of exactly zero. Given these assumptions, we can integrate the probability distribution function over the area of each region. We will use x_{min}, x_{max}, y_{min} and y_{max} to denote the edges of the region, μ_x and μ_y as the location reported by the sensor, σ as the standard deviation along either axis (for which we will determine a suitable value later) and $erf()$ is the error function. We then get the following formula [5] for the cumulative distribution function:

$$\frac{1}{4}\left(\mathrm{erf}\Big(\frac{x_{max}-\mu_x}{\sqrt{2}\,\sigma}\Big)-\mathrm{erf}\Big(\frac{x_{min}-\mu_x}{\sqrt{2}\,\sigma}\Big)\right)\left(\mathrm{erf}\Big(\frac{y_{max}-\mu_y}{\sqrt{2}\,\sigma}\Big)-\mathrm{erf}\Big(\frac{y_{min}-\mu_y}{\sqrt{2}\,\sigma}\Big)\right)$$

Using that formula on each of the cells in a grid, we get a distribution similar to what is shown in figure 4. The cell containing coordinate (μ_x, μ_y) will always have the highest total probability, in this example 0.138125. The lack of symmetry is caused by (μ_x, μ_y) not being at the centre of the cell which contains it.

Figure 4: Example of probability distribution

2,26E-06	4,44E-05	0,00034	0,001027	0,001234	0,000591	0,000112	8,34E-06	2,40E-07	2,65E-09	1,11E-11
2,77E-05	0,000543	0,004158	0,012563	0,0151	0,00723	0,001372	0,000102	2,94E-06	3,25E-08	1,36E-10
0,000133	0,002609	0,01997	0,060339	0,072525	0,034727	0,006588	0,00049	1,41E-05	1,56E-07	6,53E-10
0,000253	0,004969	0,038033	0,114916	0,138125	0,066139	0,012547	0,000933	2,69E-05	2,97E-07	1,24E-09
0,000192	0,00377	0,028862	0,087204	0,104816	0,050189	0,009522	0,000708	2,04E-05	2,25E-07	9,44E-10
5,80E-05	0,001138	0,008708	0,026311	0,031625	0,015143	0,002873	0,000214	6,16E-06	6,80E-08	2,85E-10
6,90E-06	0,000135	0,001036	0,00313	0,003762	0,001802	0,000342	2,54E-05	7,33E-07	8,09E-09	3,39E-11
3,20E-07	6,28E-06	4,80E-05	0,000145	0,000174	8,35E-05	1,58E-05	1,18E-06	3,40E-08	3,75E-10	1,57E-12
5,72E-09	1,12E-07	8,58E-07	2,59E-06	3,12E-06	1,49E-06	2,83E-07	2,11E-08	6,07E-10	6,70E-12	2,81E-14
3,89E-11	7,63E-10	5,84E-09	1,77E-08	2,12E-08	1,02E-08	1,93E-09	1,43E-10	4,13E-12	4,56E-14	1,91E-16
1,00E-13	1,96E-12	1,50E-11	4,54E-11	5,46E-11	2,62E-11	4,96E-12	3,69E-13	1,06E-14	1,17E-16	4,92E-19

We can use the approach above to calculate an evidence value for each region, which we can then use just like any other evidence value. For instance, we can perform fusion on multiple of these values to combine them into a single value. As you may recall, when using the raw sensor output, it is possible to receive multiple locations in which a person might have been detected. With this method we have a way of combining all those values together and obtain a single evidence value. We can also combine the results from both sensors together into a single result.

One thing to keep in mind is that each location returned by the sensors consists of not only an x and y coordinate, but also a weight value. Since a higher score signifies a higher confidence (and no more details than that are available), the weight is used directly as a factor by which to scale the value assigned to each region.

In section 3.2.1 we saw the various files for each data set contain their own timestamps; the values in one file are distinct from the ones in the other files. For our analysis we will generate timestamps at fixed intervals. These timestamps will, in general, not appear in any of the files. For the sensors we solve this by looking at the most recent data point prior to that timestamp. This matches the behaviour we would see when doing analysis in real time. For ground truth we perform a linear interpolation between the previous and next location recorded. While this does mean we are "peeking into the future", for ground truth this is not a problem, since it is an oracle anyway.

[5] Formula derived by A.C.C. van Wijk (personal communication, July 6th, 2018).

4.2.4 Parameter values

The model described in the previous section contains a number of parameters we have not yet determined a value for. This section will describe how we can find good values for them.

First is the grid size. There are reasons for both a coarse and a fine grid. Even in the most optimal situation, the best we can do is find the correct grid cell, but we have no way of determining where in the cell the person is, so the best we can do is report the middle of the cell. Using a coarse grid would limit the maximum accuracy we can achieve. On the other hand, a person is not a dimensionless point; if the physical person takes up many grid cells at once, it is not clear in which cell the sensor will report them. Neither do we have a way to ensure that, even if a person is standing still, they will always be reported in the same grid cell. This could have a negative effect on our attempt to collect evidence over time. Finally, using an overly-fine grid also causes an increase in the computation time needed to analyse the collected data. We decide to use a grid with 100 mm by 100 mm cells. This is not the result of an optimisation algorithm, but simply a choice which seems a reasonable compromise.

Next we look for the optimal values for the amount of evidence retained from one sample to the next. The source for this program is given in appendix B.3.2. Table 21 shows the results.

Table 21: RMS error for various evidence retention factors

Weight	Training A		Training B		Training C		Training D	
	Wall	Window	Wall	Window	Wall	Window	Wall	Window
0.1	505	494	477	473	494	645	580	497
0.2	500	490	469	467	487	645	580	491
0.3	493	484	460	460	482	647	576	482
0.4	481	478	449	455	476	647	571	475
0.5	473	473	443	449	469	650	574	468
0.6	462	461	438	435	468	668	583	453
0.7	463	455	437	426	472	692	579	442
0.8	496	490	464	449	510	766	621	461
0.9	713	686	643	620	705	1049	783	598

Unfortunately there is no single optimum which works for all data sets. However, 0.7 looks to be a good compromise; it lets most cases gain a little extra accuracy, but is below the point where inaccuracy starts to increase significantly.

Finally we investigate the best value for σ. The code in appendix B.3.2 is almost identical to the previous program. Its results are listed in table 21.

Accuracy slowly but steadily increases until about 2000. After that, some data sets keep increasing their accuracy ever so slightly, where others have the smallest of decreases in accuracy. Note this value represents millimetres, so this would mean a standard deviation of 2 metres. This seems remarkably large for a testing area of roughly 2.0 by 2.5 metres. One possible reason is that we simply select the grid cell which contains the highest amount of positive evidence; we give no penalty or bonus for "winning" by a tiny or large margin. If this explanation is correct, after reaching optimum accuracy around 2000, increasing σ further only very rarely changes which grid cell has the most evidence and the tiny differences we see in accuracy are essentially just noise.

Table 22: RMS error for various σ values

σ	Training A		Training B		Training C		Training D	
	Wall	Window	Wall	Window	Wall	Window	Wall	Window
250	462	455	436	426	472	691	579	441
500	439	422	412	388	448	685	590	402
750	422	406	396	371	434	687	574	382
1000	416	400	391	363	429	689	570	372
2000	411	391	385	354	431	693	564	363
5000	410	388	385	351	434	695	561	362
10000	410	388	385	351	434	695	561	362
20000	410	388	385	351	435	695	561	362
50000	410	388	385	351	435	695	561	363

4.2.5 Additional improvements

When we looked at fusing the results from the proprietary algorithm, one of the approaches considered used weighted averaging, based on the distance between a sensor and the detected location. A similar solution could be used here as well; before we fuse the trust values from both sensors, we can scale them. We use the same scale factors: 1 for the sensor which reported a location closest to it and the ratio of the smaller distance to the larger distance for the other sensor. We also raise this factor to different powers to see which value gives the best accuracy. Running the analysis on our training data with a number of different weight values allows us to choose which weights works best.

Table 23: RMS error for various weight values

	Training A	Training B	Training C	Training D
Proprietary				
Fused sensors (weighted average)	350.170	388.607	448.554	439.376
Fused sensors (weighted average^2)	330.939	373.806	437.108	457.881
Fused sensors (weighted average^4)	321.971	365.848	440.371	489.028
Fused sensors (weighted average^8)	432.556	469.380	534.806	567.985
Subjective logic				
Fused sensors (weighted sum)	304.726	364.874	509.249	365.039
Fused sensors (weighted sum^2)	299.899	359.384	497.404	365.277
Fused sensors (weighted sum^4)	299.026	353.005	479.862	372.988
Fused sensors (weighted sum^8)	297.785	346.868	472.602	401.972
Fused sensors (weighted sum^{16})	301.311	343.604	474.052	427.850
Fused sensors (weighted sum^{32})	304.162	345.287	475.006	439.202
Fused sensors (weighted sum^{64})	306.301	344.574	475.872	451.726

For now we only compare the weighted approaches amongst themselves. For the weighted average of the proprietary results, it seems like raising to the second power gives accurate results most consistently. When considering the weighted sum of subjective logic opinions, raising to the fourth power seems a better compromise.

4.2.6 Results

We are now ready to calculate the results of the various algorithms and sensors, by running the code in appendix B.3.1. We can make two kinds of relevant comparisons in table 24. First there is the comparison between the proprietary algorithm and the subjective logic-based algorithm using the same sensors (comparing different algorithm for different sensors is meaningless). Secondly we can compare the accuracy of either sensor by itself and see whether sensor fusion improves the accuracy.

Table 24: RMS error of most accurate algorithms

	Test A	Test B	Test C	Test D
Proprietary				
Wall sensor	539	497	499	782
Window sensor	484	455	641	439
Fused sensors (straight average)	427	393	426	444
Fused sensors (weighted average[2])	378	339	382	418
Subjective logic				
Wall sensor	459	427	432	589
Window sensor	476	434	656	423
Fused sensors (simple sum)	399	338	513	354
Fused sensors (weighted sum[4])	358	305	417	346

5 Discussion

5.1 Answering the research questions

RQ 1: Is SL sensor fusion on dissimilar sensors more accurate than a single sensor?

Despite multiple attempts, no method was found to achieve more accurate results using multiple sensors than a single sensor. Since all versions of this experiment ended up failing to achieve the desired result, it is difficult to pinpoint the exact reason why our approach did not work. Still, the answer to this question has to be that we did not succeed in finding a way to design a sensor fusion algorithm with higher accuracy than the most accurate single sensor.

Of course, this does not necessarily mean it is impossible. If a further investigation is attempted, we would suggest to first attempt another data set. If fusion is successful there, that might provide clues as to the crucial way the data sets differ or to pin down why the approach taken in this work failed to achieve the desired result.

One way to modify our approach would be to reconsider the way in which the models are optimised. When we searched for the best parameters, we only look at whether or not they got the right answer; model A and B essentially only consider whether belief in the correct answer is higher than 0.5 (while model C is slightly more complex, when optimising we still only differentiate between "correct" and "incorrect"). While this makes sense for a model which will only be used by itself, it might very well be suboptimal when the prediction of the model is used as input to a fusion operator. An optimisation strategy which not only rewards correct answers, but also gives smaller rewards for getting the answer almost right (in other words, awarding a better score to 0.40 trust in the correct prediction than to having 0.10 trust in it) might find a model which produces output more suitable as input for fusion.

RQ 2: Is SL sensor fusion more accurate than Xetal Kinsei's builtin algorithm?

For fairness' sake, it should be noted the builtin algorithm is also capable of tracking multiple people at once, something the current version of our algorithm cannot do. It is at least possible some compromises were made which benefit tracking of multiple people but hurt tracking of a single person. That makes it impossible to say whether this comparison is completely fair.

In almost all cases our accuracy is higher; sometimes by a little, sometimes by a lot. This question can therefore be answered affirmatively. Since we have no idea how the proprietary algorithm works, it is unfortunately impossible to discuss in detail *why* we manage to reach higher accuracy.

RQ 3a: Is SL sensor fusion on similar sensors more accurate than a single sensor?

Unlike occupancy detection, in this case we do manage to perform sensor fusion in such a way that the accuracy of our results is usually higher than what can be obtained from any single sensor. Except for data set Test C we outperform not only the built-in algorithm, but also our own algorithm.

Out of curiosity, we decided to have a look at the third data set in the "C" group as well. As table 25 shows, all values are very similar between these two data sets, which practically rules out an error in, or corruption of, the data files. Unfortunately we cannot determine a plausible reason why the window sensor has such poor accuracy in this setup. In particular, the only known source of inaccuracy is having one of the markers just outside of its range. It is unclear how that would hurt accuracy so much more (from the 400 – 500 range to roughly 650) than what we see for the wall sensor in Test D, for which we deliberately occluded a large part of the testing area.

Table 25: RMS error of additional data set

	Spare C	Test C
Proprietary		
Wall sensor	527	499
Window sensor	636	641
Fused sensors (straight average)	427	426
Fused sensors (weighted average[2])	379	382
Subjective logic		
Wall sensor	446	432
Window sensor	647	656
Fused sensors (simple sum)	528	513
Fused sensors (weighted sum[4])	435	417

RQ 3b: Is SL sensor fusion on similar sensors more accurate than averaging?

While the increase in accuracy is admittedly small in some test cases, we do score better in three out of four cases. Just as with RQ 3a, no satisfactory explanation has been found for the behaviour of Test C.

6 Conclusion and future work

6.1 Conclusion

We have shown that in some cases subjective logic can be a useful tool when performing sensor fusion. It is, however, by no means a silver bullet which works in all cases. A limitation is the fact it does require a little work to set things up. So far, the only way to tell whether our approach works in a given situation is to obtain a data set, develop a model for turning sensor readings into trust values and actually run the experiment. A benefit is that the underlying mathematics is very straightforward and easy to implement, even when tools such as Matlab and libraries of advanced mathematical functions are not available. The most complicated mathematics necessary for the subjective logic analysis itself is literally floating point exponentiation [6], which makes this approach suitable for implementation on just about any device.

6.2 Future work

In the location detection case, test C performed very poorly compared to the other data sets we looked at. It is not at all clear what causes this strange behaviour. Taking the time to investigate what is going on exactly could very well lead to interesting new insights.

The most interesting question which remains unanswered for now is how we can determine (if possible, a priori, only using a description of the intended use case) whether our approach is likely to work in a given situation, without having to fully implement and test it. In our case, a lot of time went into the occupancy detection scenario, before it became clear it was not going to work.

While developing the location detection model, one of our steps was determining an optimal value for σ. Contrary to expectations, it appears such an optimum does not really exist. While we do have a hypothesis about a possible cause, it would be interesting to explore further and see what is going on exactly.

[6] In the case of *location detection*, running the full analysis also requires an implementation of the error function.

References

[1] Kinsei product information page. https://www.xetal.eu/kinsei.

[2] Vinh Bui, Richard Verhoeven, and Johan Lukkien. Evaluating trustworthiness through monitoring: The foot, the horse and the elephant. In T. Holz and S. Ioannidis, editors, *Trust and Trustworthy Computing (7th International Conference, TRUST 2014, Heraklion, Crete, June 30-July 2, 2014. Proceedings)*, Lecture Notes in Computer Science, pages 188–205, Germany, 2014. Springer.

[3] Luis M. Candanedo and Véronique Feldheim. Accurate occupancy detection of an office room from light, temperature, humidity and co2 measurements using statistical learning models. *Energy and Buildings*, 112:28 – 39, 2016.

[4] Cicek Guven, Mike Holenderski, Tanir Ozcelebi, and Johan Lukkien. A formalization of computational trust. In *Joint 13th CTTE and 10th CMI Conference on Internet of Things - Business Models, Users, and Networks*, volume 2018-January, United States, 1 2018. Institute of Electrical and Electronics Engineers (IEEE).

[5] Ebenezer Hailemariam, Rhys Goldstein, Ramtin Attar, and Azam Khan. Real-time occupancy detection using decision trees with multiple sensor types. In *Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design*, SimAUD '11, pages 141–148, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[6] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, August 2001.

[7] Audun Jøsang. *Subjective Logic - A Formalism for Reasoning Under Uncertainty*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.

[8] Timilehin Labeodan, Wim Zeiler, Gert Boxem, and Yang Zhao. Occupancy measurement in commercial office buildings for demand-driven control applicationsa survey and detection system evaluation. *Energy and Buildings*, 93:303 – 314, 2015.

[9] Christian Martin, Erik Schaffernicht, Andrea Scheidig, and Horst-Michael Gross. Multi-modal sensor fusion using a probabilistic aggregation scheme for people detection and tracking. *Robotics and Autonomous Systems*, 54(9):721 – 728, 2006. Selected papers from the 2nd European Conference on Mobile Robots (ECMR 05).

[10] M. Mousavi Moaiied and Mohammad-Reza Mosavi. Increasing accuracy of combined gps and glonass positioning using fuzzy kalman filter. *Iranian Journal of Electrical and Electronic Engineering*, 12(1), 2016.

[11] Magnus Moglia, Ashok Kumar Sharma, and Shiroma Maheepala. Multi-criteria decision assessments using subjective logic: Methodology and the case of urban water strategies. *Journal of Hydrology*, 452-453:180 – 189, 2012.

[12] Helen Svensson and Audun Jøsang. Correlation of intrusion alarms with subjective logic. 2001.

[13] Rens Wouter van der Heijden, Henning Kopp, and Frank Kargl. Multi-source fusion operations in subjective logic. *CoRR*, abs/1805.01388, 2018.

[14] Yong Wang, Huihua Yang, Xingyu Wang, and Ruixia Zhang. Distributed intrusion detection system based on data fusion method. In *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*, volume 5, pages 4331–4334 Vol.5, June 2004.

[15] Torsten Wilhelm, H.-J. Boehme, and Horst-Michael Gross. Sensor fusion for vision and sonar based people tracking on a mobile service robot. 2002.

[16] Junjing Yang, Mattheos Santamouris, and Siew Eang Lee. Review of occupancy sensing systems and occupancy modeling methodologies for the application in institutional buildings. *Energy and Buildings*, 121:344 – 349, 2016.

[17] Zheng Yang, Nan Li, Burcin Becerik-Gerber, and Michael Orosz. A systematic approach to occupancy modeling in ambient sensor-rich buildings. *SIMULATION*, 90(8):960–977, 2014.

[18] Pengfei Zhang, Rui Tu, Rui Zhang, Yuping Gao, and Hongbin Cai. Combining gps, beidou, and galileo satellite systems for time and frequency transfer based on carrier phase observations. *Remote Sensing*, 10(2), 2018.

A Results

A.1 Occupancy detection

A.1.1 Model A

Table 26: Model A, all results, using all sensors, sorted by sensors used

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	–	Used	0.737
–	–	–	Used	–	0.981
–	–	–	Used	Used	0.873
–	–	Used	–	–	0.526
–	–	Used	–	Used	0.737
–	–	Used	Used	–	0.878
–	–	Used	Used	Used	0.865
–	Used	–	–	–	0.396
–	Used	–	–	Used	0.408
–	Used	–	Used	–	0.640
–	Used	–	Used	Used	0.655
–	Used	Used	–	–	0.396
–	Used	Used	–	Used	0.410
–	Used	Used	Used	–	0.641
–	Used	Used	Used	Used	0.658
Used	–	–	–	–	0.455
Used	–	–	–	Used	0.628
Used	–	–	Used	–	0.923
Used	–	–	Used	Used	0.873
Used	–	Used	–	–	0.643
Used	–	Used	–	Used	0.742
Used	–	Used	Used	–	0.902
Used	–	Used	Used	Used	0.864
Used	Used	–	–	–	0.403
Used	Used	–	–	Used	0.412
Used	Used	–	Used	–	0.651
Used	Used	–	Used	Used	0.660
Used	Used	Used	–	–	0.402
Used	Used	Used	–	Used	0.414
Used	Used	Used	Used	–	0.650
Used	Used	Used	Used	Used	0.660

Table 27: Model A, all results, using all sensors, sorted by F-score

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.981
Used	–	–	Used	–	0.923
Used	–	Used	Used	–	0.902
–	–	Used	Used	–	0.878
Used	–	–	Used	Used	0.873
–	–	–	Used	Used	0.873
–	–	Used	Used	Used	0.865
Used	–	Used	Used	Used	0.864
Used	–	Used	–	Used	0.742
–	–	–	–	Used	0.737
–	–	Used	–	Used	0.737
Used	Used	–	Used	Used	0.660
Used	Used	Used	Used	Used	0.660
–	Used	Used	Used	Used	0.658
–	Used	–	Used	Used	0.655
Used	Used	–	Used	–	0.651
Used	Used	Used	Used	–	0.650
Used	–	Used	–	–	0.643
–	Used	Used	Used	–	0.641
–	Used	–	Used	–	0.640
Used	–	–	–	Used	0.628
–	–	Used	–	–	0.526
Used	–	–	–	–	0.455
Used	Used	Used	–	Used	0.414
Used	Used	–	–	Used	0.412
–	Used	Used	–	Used	0.410
–	Used	–	–	Used	0.408
Used	Used	–	–	–	0.403
Used	Used	Used	–	–	0.402
–	Used	Used	–	–	0.396
–	Used	–	–	–	0.396

Table 28: Model A, all results, light sensor omitted, sorted by F-score

CO2	Humidity	Humidity ratio	Temperature	F-score
Used	–	Used	Used	0.742
–	–	–	Used	0.737
–	–	Used	Used	0.737
Used	–	Used	–	0.643
Used	–	–	Used	0.628
–	–	Used	–	0.526
Used	–	–	–	0.455
Used	Used	Used	Used	0.414
Used	Used	–	Used	0.412
–	Used	Used	Used	0.410
–	Used	–	Used	0.408
Used	Used	–	–	0.403
Used	Used	Used	–	0.402
–	Used	Used	–	0.396
–	Used	–	–	0.396

Table 29: Model A, all results, light and temperature sensors omitted, sorted by F-score

CO2	Humidity	Humidity ratio	F-score
Used	–	Used	0.643
–	–	Used	0.526
Used	–	–	0.455
Used	Used	–	0.403
Used	Used	Used	0.402
–	Used	Used	0.396
–	Used	–	0.396

A.1.2 Model B

Table 30: Model B, all results, using all sensors, sorted by sensors used

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	–	Used	0.737
–	–	–	Used	–	0.981
–	–	–	Used	Used	0.874
–	–	Used	–	–	0.508
–	–	Used	–	Used	0.735
–	–	Used	Used	–	0.863
–	–	Used	Used	Used	0.860
–	Used	–	–	–	0.395
–	Used	–	–	Used	0.406
–	Used	–	Used	–	0.641
–	Used	–	Used	Used	0.655
–	Used	Used	–	–	0.394
–	Used	Used	–	Used	0.408
–	Used	Used	Used	–	0.639
–	Used	Used	Used	Used	0.656
Used	–	–	–	–	0.450
Used	–	–	–	Used	0.612
Used	–	–	Used	–	0.924
Used	–	–	Used	Used	0.873
Used	–	Used	–	–	0.586
Used	–	Used	–	Used	0.724
Used	–	Used	Used	–	0.907
Used	–	Used	Used	Used	0.866
Used	Used	–	–	–	0.402
Used	Used	–	–	Used	0.412
Used	Used	–	Used	–	0.650
Used	Used	–	Used	Used	0.660
Used	Used	Used	–	–	0.401
Used	Used	Used	–	Used	0.412
Used	Used	Used	Used	–	0.648
Used	Used	Used	Used	Used	0.660

Table 31: Model B, all results, using all sensors, sorted by F-score

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.981
Used	–	–	Used	–	0.924
Used	–	Used	Used	–	0.907
–	–	–	Used	Used	0.874
Used	–	–	Used	Used	0.873
Used	–	Used	Used	Used	0.866
–	–	Used	Used	–	0.863
–	–	Used	Used	Used	0.860
–	–	–	–	Used	0.737
–	–	Used	–	Used	0.735
Used	–	Used	–	Used	0.724
Used	Used	Used	Used	Used	0.660
Used	Used	–	Used	Used	0.660
–	Used	Used	Used	Used	0.656
–	Used	–	Used	Used	0.655
Used	Used	–	Used	–	0.650
Used	Used	Used	Used	–	0.648
–	Used	–	Used	–	0.641
–	Used	Used	Used	–	0.639
Used	–	–	–	Used	0.612
Used	–	Used	–	–	0.586
–	–	Used	–	–	0.508
Used	–	–	–	–	0.450
Used	Used	Used	–	Used	0.412
Used	Used	–	–	Used	0.412
–	Used	Used	–	Used	0.408
–	Used	–	–	Used	0.406
Used	Used	–	–	–	0.402
Used	Used	Used	–	–	0.401
–	Used	–	–	–	0.395
–	Used	Used	–	–	0.394

Table 32: Model B, all results, light sensor omitted, sorted by F-score

CO2	Humidity	Humidity ratio	Temperature	F-score
–	–	–	Used	0.737
–	–	Used	Used	0.735
Used	–	Used	Used	0.724
Used	–	–	Used	0.612
Used	–	Used	–	0.586
–	–	Used	–	0.508
Used	–	–	–	0.450
Used	Used	Used	Used	0.412
Used	Used	–	Used	0.412
–	Used	Used	Used	0.408
–	Used	–	Used	0.406
Used	Used	–	–	0.402
Used	Used	Used	–	0.401
–	Used	–	–	0.395
–	Used	Used	–	0.394

Table 33: Model B, all results, light and temperature sensors omitted, sorted by F-score

CO2	Humidity	Humidity ratio	F-score
Used	–	Used	0.586
–	–	Used	0.508
Used	–	–	0.450
Used	Used	–	0.402
Used	Used	Used	0.401
–	Used	–	0.395
–	Used	Used	0.394

A.1.3 Model C

Table 34: Model C, all results, using all sensors, sorted by sensors used

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	–	Used	0.745
–	–	–	Used	–	0.982
–	–	–	Used	Used	0.907
–	–	Used	–	–	0.524
–	–	Used	–	Used	0.665
–	–	Used	Used	–	0.952
–	–	Used	Used	Used	0.922
–	Used	–	–	–	0.354
–	Used	–	–	Used	0.375
–	Used	–	Used	–	0.748
–	Used	–	Used	Used	0.741
–	Used	Used	–	–	0.369
–	Used	Used	–	Used	0.397
–	Used	Used	Used	–	0.743
–	Used	Used	Used	Used	0.726
Used	–	–	–	–	0.459
Used	–	–	–	Used	0.682
Used	–	–	Used	–	0.968
Used	–	–	Used	Used	0.913
Used	–	Used	–	–	0.497
Used	–	Used	–	Used	0.744
Used	–	Used	Used	–	0.924
Used	–	Used	Used	Used	0.905
Used	Used	–	–	–	0.357
Used	Used	–	–	Used	0.380
Used	Used	–	Used	–	0.743
Used	Used	–	Used	Used	0.706
Used	Used	Used	–	–	0.365
Used	Used	Used	–	Used	0.371
Used	Used	Used	Used	–	0.726
Used	Used	Used	Used	Used	0.728

Table 35: Model C, all results, using all sensors, sorted by F-score

CO2	Humidity	Humidity ratio	Light	Temperature	F-score
–	–	–	Used	–	0.982
Used	–	–	Used	–	0.968
–	–	Used	Used	–	0.952
Used	–	Used	Used	–	0.924
–	–	Used	Used	Used	0.922
Used	–	–	Used	Used	0.913
–	–	–	Used	Used	0.907
Used	–	Used	Used	Used	0.905
–	Used	–	Used	–	0.748
–	–	–	–	Used	0.745
Used	–	Used	–	Used	0.744
–	Used	Used	Used	–	0.743
Used	Used	–	Used	–	0.743
–	Used	–	Used	Used	0.741
Used	Used	Used	Used	Used	0.728
–	Used	Used	Used	Used	0.726
Used	Used	Used	Used	–	0.726
Used	Used	–	Used	Used	0.706
Used	–	–	–	Used	0.682
–	–	Used	–	Used	0.665
–	–	Used	–	–	0.524
Used	–	Used	–	–	0.497
Used	–	–	–	–	0.459
–	Used	Used	–	Used	0.397
Used	Used	–	–	Used	0.380
–	Used	–	–	Used	0.375
Used	Used	Used	–	Used	0.371
–	Used	Used	–	–	0.369
Used	Used	Used	–	–	0.365
Used	Used	–	–	–	0.357
–	Used	–	–	–	0.354

Table 36: Model C, all results, light sensor omitted, sorted by F-score

CO2	Humidity	Humidity ratio	Temperature	F-score
–	–	–	Used	0.745
Used	–	Used	Used	0.744
Used	–	–	Used	0.682
–	–	Used	Used	0.665
–	–	Used	–	0.524
Used	–	Used	–	0.497
Used	–	–	–	0.459
–	Used	Used	Used	0.397
Used	Used	–	Used	0.380
–	Used	–	Used	0.375
Used	Used	Used	Used	0.371
–	Used	Used	–	0.369
Used	Used	Used	–	0.365
Used	Used	–	–	0.357
–	Used	–	–	0.354

Table 37: Model C, all results, light and temperature sensors omitted, sorted by F-score

CO2	Humidity	Humidity ratio	F-score
–	–	Used	0.524
Used	–	Used	0.497
Used	–	–	0.459
–	Used	Used	0.369
Used	Used	Used	0.365
Used	Used	–	0.357
–	Used	–	0.354

B Source code

B.1 Library code

B.1.1 sl.py - Subjective logic

```
from typing import Dict, List

import time
import csv


class Evidence:
    # For r, s, w: see Josang, "Subjective logic" Book draft (2011)
    # http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.5947&rep=rep1&type=pdf
    #
    # For gamma, delta, zeta: see Bui, Verhoeven and Lukkien, "Evaluating trustworthiness through
    # monitoring: The foot, the horse and the elephant." International Conference on Trust and
    # Trustworthy Computing, Springer (2014)
    #
    # r: number of observations in favour of trust statement (amount of evidence in favour)
    # s: number of observations against trust statement (amount of evidence against)
    # w: small constant, also prevents division by zero when r = s = 0
    #
    # gamma: fraction of evidence against to retain for observation in favour
    # delta: fraction of evidence in favour to retain for observation against
    # zeta:  fraction of evidence to retain for missing observation
    #
    # Leave Greek parameters at defaults to tally r and s as simple counters
    def __init__(self,
                 r: int = 0,
                 s: int = 0,
                 w: int = 2,
                 gamma: float = 1,
                 delta: float = 1,
                 zeta: float = 1):
        if r < 0:
            raise ValueError (
                'ERROR: parameter r given as {r:.4f}, but must be non-negative'.format ( r = r )
            )
        if s < 0:
            raise ValueError (
                'ERROR: parameter s given as {s:.4f}, but must be non-negative'.format ( s = s )
            )
        if w < 0:
            raise ValueError (
                'ERROR: parameter w given as {w:.4f}, but must be non-negative'.format ( w = w )
            )

        if ( gamma < 0 ) or ( 1 < gamma ):
            raise ValueError (
                'ERROR: parameter gamma given as {gamma:.4f}, but must be in range [0 ; 1]'.format (
                    gamma = gamma
                )
            )
        if ( delta < 0 ) or ( 1 < delta ):
            raise ValueError (
                'ERROR: parameter delta given as {delta:.4f}, but must be in range [0 ; 1]'.format (
                    delta = delta
                )
            )
        if ( zeta < 0 ) or ( 1 < zeta ):
            raise ValueError (
                'ERROR: parameter zeta given as {delta:.4f}, but must be in range [0 ; 1]'.format (
                    zeta = zeta
                )
            )

        self.r = r
        self.s = s
        self.w = w

        self.gamma = gamma
        self.delta = delta
        self.zeta = zeta


    # Add an observation
    # -1             missing observation
    # Range [0; 1]   observed value
    def observe(self, obs: float):
        # Known issue: the logic of this function compares a float for equality
        # Rationale: while values in the [0 ; 1] range could be the result of a computation (and
        #            therefore suffer from inexactness), a -1 should always be assigned as literal
        #            and be exact.

        if ( obs != -1 ) and ( ( obs < 0 ) or ( 1 < obs ) ):
            raise ValueError (
                'ERROR: observation given as {obs:.4f}, but must be -1 or in range [0 ; 1]'.format (
                    obs = obs
                )
            )

        if obs == -1:
            self.r = self.zeta * self.r
            self.s = self.zeta * self.s
```

```
        else:
            self.r = ( self.delta ** ( 1 - obs ) ) * self.r + obs
            self.s = ( self.gamma ** obs ) * self.s + 1 - obs


    def get_r(self):
        return self.r


    def get_s(self):
        return self.s


    def get_w(self):
        return self.w


    def get_belief(self):
        return self.r / ( self.r + self.s + self.w )


    def get_disbelief(self):
        return self.s / ( self.r + self.s + self.w )


    def get_uncert(self):
        return self.w / ( self.r + self.s + self.w )


    def get_conf(self):
        return 1 - self.get_uncert()



class Opinion:
    # domain: the names of the domain elements; used as keys for belief and base rate values
    # belief: amount of belief in each given domain element (omitted values set to zero)
    # base: base rate of each given domain element (if given values sum to less than one,
    #       remainder is shared equally among elements not explicitly given)
    # uncert: amount of belief not assigned to any domain element (automatically calculated
    #         if omitted; used as sanity check when provided)
    def __init__(self,
                 domain: List[str],
                 belief: Dict[str, float],
                 base: Dict[str, float] = {},
                 uncert: float = None):
        remaining_base = 1 - sum ( base.values() )
        default_bases = len ( domain ) - len ( base )

        for x in domain:
            if x not in belief:
                belief[x] = 0.0

            if x not in base:
                base[x] = remaining_base / default_bases

        if uncert == None:
            uncert = 1 - sum ( belief.values() )

        self.domain = domain
        self.belief = belief
        self.uncert = uncert
        self.base = base

        self._check()


    # Perform internal consistency checks.
    def _check(self):
        error = None

        if abs ( sum ( self.belief.values() ) + self.uncert - 1 ) > 0.0001:
            error = 'ERROR: total belief mass unequal to 1'

        if abs ( sum ( self.base.values() ) - 1 ) > 0.0001:
            error = 'ERROR: total base rate unequal to 1'

        for x in self.domain:
            if self.belief[x] < 0.0:
                error = 'ERROR: negative belief mass'

            if self.base[x] < 0.0:
                error = 'ERROR: negative base rate'

        if error != None:
            print ( '\n\n-- ERROR --- Opinion object failed internal consistency checks:' )
            self.print()
            raise Exception ( error )



# Fuse two Opinions into a single Opinion
# See https://folk.uio.no/josang/sl/
# "Subjective Logic - A Formalism for Reasoning Under Uncertainty"
# for explanation of calculations
def cumulative_fusion(a: Opinion, b: Opinion):
    if a.domain != b.domain:
        raise Exception ( 'Cannot fuse values with different domains' )
```

```
    domain = a.domain
    belief = {}
    base = {}

    for x in domain:
        if a.uncert == 0 and b.uncert == 0:
            belief[x] = ( a.belief[x] + b.belief[x] ) / 2
        else:
            belief[x] = ( a.belief[x] * b.uncert + b.belief[x] * a.uncert ) /
                        ( a.uncert + b.uncert - a.uncert * b.uncert )

        if a.uncert == 1 and b.uncert == 1:
            base[x] = ( a.base[x] + b.base[x] ) / 2
        else:
            base[x] = ( a.base[x] * b.uncert + b.base[x] * a.uncert -
                        ( a.base[x] + b.base[x] ) * a.uncert * b.uncert ) /
                      ( a.uncert + b.uncert - 2 * a.uncert * b.uncert )

    if a.uncert == 0 and b.uncert == 0:
        uncert = 0
    else:
        uncert = ( a.uncert * b.uncert ) / ( a.uncert + b.uncert - a.uncert * b.uncert )

    result = Opinion (
                domain = domain,
                belief = belief,
                base = base,
                uncert = uncert
            )

    result._check()

    return result
```

B.1.2 delayed_csv.py - Parse location sensor data

```
import csv
import datetime
import json


class DelayedCsv:
    def __init__(self, filename, delay):
        ## Analyse input file
        file = open ( filename )
        reader = csv.reader ( file, delimiter = '\t', quotechar = '"', skipinitialspace = True )

        line = []
        while len ( line ) == 0 or line[0][0:25] != 'Starting_persons_tracking':
            line = next ( reader )

        date = line[0][-10:]

        line = next ( reader )
        time_start = line[0]

        while line[0][0:17] != 'Finished_tracking':
            time_stop = line[0]
            line = next ( reader )

        file.close()

        ## Actual init
        self._file = open ( filename )
        self._reader = csv.reader ( self._file, delimiter = '\t', quotechar = '"', skipinitialspace = True )

        line = []
        while len ( line ) == 0 or line[0][0:25] != 'Starting_persons_tracking':
            line = next ( self._reader )

        self._epoch = datetime.datetime ( 1970, 1, 1, 0, 0, 0 )
        self._date = date
        self._delay = delay
        self.time_start = self._time_str_to_float ( time_start )
        self.time_stop = self._time_str_to_float ( time_stop )

        self._prev_line = None
        self._next_line = None
        self._advance()
        self._advance()


    def _advance(self):
        line = next ( self._reader )

        # In theory, the first index might be empty, with a non-empty result in the second index.
        # Since this corner case does not occur in the collected data, it is ignored here.
        fused = json.loads ( line [3] )[0]
        raw = json.loads ( line [4] )

        # In theory, the first line might contain an empty result, which would break this code.
        # Since this corner case does not occur in the collected data, it is ignored here.
        if fused == []:
            latest_location = self._next_line['latest_location']
        else:
```

```
            latest_location = fused

        line = { 'time': self._time_str_to_float ( line[0] ),
                 'detections': raw,
                 'location': fused, 'latest_location': latest_location }

        self._prev_line = self._next_line
        self._next_line = line


    def _time_str_to_float(self, time: str):
        timestamp = datetime.datetime.strptime (
            '{date} {time}'.format ( date = self._date, time = time ), '%Y-%m-%d %H:%M:%S.%f'
        )
        timestamp = timestamp + datetime.timedelta ( seconds = self._delay)
        return ( timestamp - self._epoch ).total_seconds()


    def get_detections(self, time: float):
        while time > self._next_line['time']:
            self._advance()

        result = []

        for coord in self._prev_line['detections']:
            result.append ( { 'x': coord[0], 'y': coord[1], 'weight': coord[2] } )

        return result


    def get_location(self, time: float, allow_none: bool = True):
        while time > self._next_line['time']:
            self._advance()

        if allow_none:
            coord = self._prev_line['location']
        else:
            coord = self._prev_line['latest_location']

        if coord == []:
            result = None
        else:
            result = { 'x': coord[0], 'y': coord[1] }

        return result
```

B.1.3 interpolated_csv.py - Parse location ground truth data

```
import csv
import datetime


# Locations for 20180728 experiments; __init__() makes correction when opening a datafile for 20180729
LOCATIONS = {
    'A': { 'x': 2600, 'y': 3200 },
    'B': { 'x': 3800, 'y': 3200 },
    'C': { 'x': 5000, 'y': 3200 },
    'D': { 'x': 3200, 'y': 2200 },
    'E': { 'x': 4400, 'y': 2200 },
    'F': { 'x': 2600, 'y': 1200 },
    'G': { 'x': 3800, 'y': 1200 },
    'H': { 'x': 5000, 'y': 1200 },
}


class InterpolatedCsv:
    def __init__(self, filename ):
        ## Analyse input file
        file = open ( filename )
        reader = csv.reader ( file, delimiter = ' ', quotechar = '"', skipinitialspace = True )

        line = next ( reader )
        date = line[0]
        time_start = line[1]

        try:
            while True:
                line = next ( reader )
        except StopIteration:
            pass
        time_stop = line[1]
        file.close()

        ## One location was moved on the second day of experiments; patch up the 'constant' if necessary
        if date == '2018-07-29':
            LOCATIONS['C']['y'] = 2800


        ## Actual init
        self._file = open ( filename )
        self._reader = csv.reader ( self._file, delimiter = ' ', quotechar = '"', skipinitialspace = True )

        self._epoch = datetime.datetime ( 1970, 1, 1, 0, 0, 0 )
        self._date = date
        self.time_start = self._time_str_to_float ( time_start )
```

```
        self.time_stop = self._time_str_to_float ( time_stop )

        self._prev_line = None
        self._next_line = None
        self._advance()
        self._advance()


    def _advance(self):
        line = next ( self._reader )
        line = { 'time': self._time_str_to_float ( line[1] ), 'location': line[2] }

        self._prev_line = self._next_line
        self._next_line = line


    def _time_str_to_float(self, time: str):
        timestamp = datetime.datetime.strptime (
            '{date} {time}'.format ( date = self._date, time = time ), '%Y-%m-%d %H:%M:%S.%f'
        )
        return ( timestamp - self._epoch ).total_seconds()


    def get(self, time: float):
        while time > self._next_line['time']:
            self._advance()

        time_prev = self._prev_line['time']
        time_next = self._next_line['time']

        ratio = ( time - time_prev ) / ( time_next - time_prev )

        coord_prev = LOCATIONS[self._prev_line['location']]
        coord_next = LOCATIONS[self._next_line['location']]

        x = coord_prev['x'] + ratio * ( coord_next['x'] - coord_prev['x'] )
        y = coord_prev['y'] + ratio * ( coord_next['y'] - coord_prev['y'] )

        return { 'x': x, 'y': y }
```

B.1.4 pm.py - Performance Measurement

```
from typing import Dict, List


class PerfMeasure:
    def __init__(self):
        self.fn = 0
        self.tn = 0
        self.fp = 0
        self.tp = 0


    def observe(self, expected, actual):
        if expected == 0 and actual == 0:
            self.tn = self.tn + 1
        elif expected == 0 and actual == 1:
            self.fp = self.fp + 1
        elif expected == 1 and actual == 0:
            self.fn = self.fn + 1
        elif expected == 1 and actual == 1:
            self.tp = self.tp + 1
        else:
            raise Exception ( 'unsupported_values' )


    def get_accuracy(self):
        try:
            return ( self.tn + self.tp ) / ( self.fn + self.fp + self.tn + self.tp )
        except:
            return -1


    def get_precision(self):
        try:
            return self.tp / ( self.fp + self.tp )
        except:
            return -1


    def get_recall(self):
        try:
            return self.tp / ( self.fn + self.tp )
        except:
            return -1


    def get_f1(self):
        try:
            return 2 * ( self.get_recall() * self.get_precision() ) /
                   ( self.get_recall() + self.get_precision() )
        except:
            return -1
```

B.1.5 sensor.py - Occupancy sensors

```
class Sensor:
    def __init__(self, lower_bound: float, upper_bound: float, rate: float):
        if lower_bound >= upper_bound:
            raise ValueError ( 'ERROR: lower_bound must be smaller than upper_bound' )
        if rate < 0:
            raise ValueError ( 'ERROR: rate-of-change must be non-negative' )

        self._last_value = None

        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
        self.rate = rate


    def convert(self, value: float):
        if self._last_value == None:
            diff = -1           # If there is no last value, we cannot calculate a diff.
                                # Since rate is forced to be non-negative in __init__ this
                                # is guaranteed to fail the diff > rate test and use the
                                # branch which relies solely on the current value.
        else:
            diff = abs ( value - self._last_value )


        if diff > self.rate:
            if value < self._last_value:
                result = 0.0
            else:
                result = 1.0
        else:
            if value < self.lower_bound:
                result = 0.0
            elif value > self.upper_bound:
                result = 1.0
            else:
                result = ( value - self.lower_bound ) / ( self.upper_bound - self.lower_bound )

        self._last_value = value

        return result
```

B.2 Occupancy detection

B.2.1 Analyse model A

```
import csv
import time
import sys

from SubjectiveLogic import sl
from Tools import sensor
from Tools import pm

#DATA_FILE = 'data\\occupancy\\test.csv'
DATA_FILE = 'data\\occupancy\\train.csv'
#DATA_FILE = 'data\\occupancy\\validate.csv'


def run_test(use_co2, use_humidity, use_humratio, use_light, use_temperature):
    evidence_co2         = sl.Evidence ( gamma = 0.83, delta = 0.90 )
    evidence_humidity    = sl.Evidence ( gamma = 0.32, delta = 1.00 )
    evidence_humratio    = sl.Evidence ( gamma = 0.91, delta = 0.93 )
    evidence_light       = sl.Evidence ( gamma = 0.18, delta = 0.48 )
    evidence_temperature = sl.Evidence ( gamma = 0.90, delta = 0.90 )

    sensor_co2         = sensor.Sensor ( lower_bound =  410.0      , upper_bound = 1010.0      , rate =   3.000    )
    sensor_humidity    = sensor.Sensor ( lower_bound =   38.7      , upper_bound =   40.7      , rate =   0.160    )
    sensor_humratio    = sensor.Sensor ( lower_bound =    0.004270, upper_bound =    0.006070, rate =   0.000001 )
    sensor_light       = sensor.Sensor ( lower_bound =  200.0      , upper_bound =  600.0      , rate = 100.000    )
    sensor_temperature = sensor.Sensor ( lower_bound =   19.2      , upper_bound =   23.2      , rate =   0.008    )

    domain = [ 'empty', 'occupied' ]
    perf = pm.PerfMeasure()

    with open ( DATA_FILE ) as csvfile:
        csvreader = csv.DictReader ( csvfile, delimiter = ',', quotechar = '"', skipinitialspace = True )
        for row in csvreader:
            evidence_co2.observe         ( sensor_co2.convert         ( float ( row['CO2']           ) ) )
            evidence_humidity.observe    ( sensor_humidity.convert    ( float ( row['Humidity']      ) ) )
            evidence_humratio.observe    ( sensor_humratio.convert    ( float ( row['HumidityRatio'] ) ) )
            evidence_light.observe       ( sensor_light.convert       ( float ( row['Light']         ) ) )
            evidence_temperature.observe ( sensor_temperature.convert ( float ( row['Temperature']   ) ) )

            fused = sl.Opinion (
                domain = domain,
                belief = { 'empty': 0, 'occupied': 0 },
            )

            if use_co2:
                fused = sl.cumulative_fusion (
                    fused,
```

```
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_co2.get_disbelief(),
                                   'occupied': evidence_co2.get_belief() },
                    )
                )

            if use_humidity:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_humidity.get_disbelief(),
                                   'occupied': evidence_humidity.get_belief() },
                    )
                )

            if use_humratio:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_humratio.get_disbelief(),
                                   'occupied': evidence_humratio.get_belief() },
                    )
                )

            if use_light:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_light.get_disbelief(),
                                   'occupied': evidence_light.get_belief() },
                    )
                )

            if use_temperature:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_temperature.get_disbelief(),
                                   'occupied': evidence_temperature.get_belief() },
                    )
                )

            occupancy = int(row['Occupancy'])

            if fused.belief['occupied'] > fused.belief['empty']:
                perf.observe ( occupancy, 1 )
            else:
                perf.observe ( occupancy, 0 )


    labels = { False: '--', True: 'Used' }
    print ( '{i0:10.8s}__{i1:10.8s}__{i2:10.8s}__{i3:10.8s}__{i4:10.8s}_______{f1:6.3f}'.format (
        i0 = labels[use_co2],
        i1 = labels[use_humidity],
        i2 = labels[use_humratio],
        i3 = labels[use_light],
        i4 = labels[use_temperature],

        f1 = perf.get_f1(),
    ) )


print ( '' )
print ( 'Analysing:', DATA_FILE )
print ( '' )
print ( 'CO2__________Humidity____Hum._Ratio__Light________Temperature_____f1' )

for i0 in [False, True]:
    for i1 in [False, True]:
        for i2 in [False, True]:
            for i3 in [False, True]:
                for i4 in [False, True]:
                    run_test ( i0, i1, i2, i3, i4 )
```

B.2.2 Analyse model B

```
import csv
import time
import sys

from SubjectiveLogic import sl
from Tools import sensor
from Tools import pm

#DATA_FILE = 'data\\occupancy\\test.csv'
DATA_FILE = 'data\\occupancy\\train.csv'
#DATA_FILE = 'data\\occupancy\\validate.csv'

def run_test(use_co2, use_humidity, use_humratio, use_light, use_temperature):
```

```
evidence_co2         = sl.Evidence ( gamma = 0.80, delta = 0.90 )
evidence_humidity    = sl.Evidence ( gamma = 0.30, delta = 1.00 )
evidence_humratio    = sl.Evidence ( gamma = 0.90, delta = 0.90 )
evidence_light       = sl.Evidence ( gamma = 0.20, delta = 0.50 )
evidence_temperature = sl.Evidence ( gamma = 0.90, delta = 0.90 )

sensor_co2         = sensor.Sensor ( lower_bound =  400.0   , upper_bound = 1000.0   , rate =    3.0      )
sensor_humidity    = sensor.Sensor ( lower_bound =   39.0   , upper_bound =   40.0   , rate =    0.16     )
sensor_humratio    = sensor.Sensor ( lower_bound =    0.004, upper_bound =    0.006, rate =    0.000001 )
sensor_light       = sensor.Sensor ( lower_bound =  200.0   , upper_bound =  600.0   , rate =  100.0      )
sensor_temperature = sensor.Sensor ( lower_bound =   19.2   , upper_bound =   23.2   , rate =    0.008    )

domain = [ 'empty', 'occupied' ]
perf = pm.PerfMeasure()

with open ( DATA_FILE ) as csvfile:
    csvreader = csv.DictReader ( csvfile, delimiter = ',', quotechar = '"', skipinitialspace = True )
    for row in csvreader:
        evidence_co2.observe         ( sensor_co2.convert         ( float ( row['CO2']           ) ) )
        evidence_humidity.observe    ( sensor_humidity.convert    ( float ( row['Humidity']      ) ) )
        evidence_humratio.observe    ( sensor_humratio.convert    ( float ( row['HumidityRatio'] ) ) )
        evidence_light.observe       ( sensor_light.convert       ( float ( row['Light']         ) ) )
        evidence_temperature.observe ( sensor_temperature.convert ( float ( row['Temperature']   ) ) )

        fused = sl.Opinion (
            domain = domain,
            belief = { 'empty': 0, 'occupied': 0 },
        )

        if use_co2:
            fused = sl.cumulative_fusion (
                fused,
                sl.Opinion (
                    domain = domain,
                    belief = { 'empty': evidence_co2.get_disbelief(),
                               'occupied': evidence_co2.get_belief() },
                )
            )

        if use_humidity:
            fused = sl.cumulative_fusion (
                fused,
                sl.Opinion (
                    domain = domain,
                    belief = { 'empty': evidence_humidity.get_disbelief(),
                               'occupied': evidence_humidity.get_belief() },
                )
            )

        if use_humratio:
            fused = sl.cumulative_fusion (
                fused,
                sl.Opinion (
                    domain = domain,
                    belief = { 'empty': evidence_humratio.get_disbelief(),
                               'occupied': evidence_humratio.get_belief() },
                )
            )

        if use_light:
            fused = sl.cumulative_fusion (
                fused,
                sl.Opinion (
                    domain = domain,
                    belief = { 'empty': evidence_light.get_disbelief(),
                               'occupied': evidence_light.get_belief() },
                )
            )

        if use_temperature:
            fused = sl.cumulative_fusion (
                fused,
                sl.Opinion (
                    domain = domain,
                    belief = { 'empty': evidence_temperature.get_disbelief(),
                               'occupied': evidence_temperature.get_belief() },
                )
            )

        occupancy = int(row['Occupancy'])

        if fused.belief['occupied'] > fused.belief['empty']:
            perf.observe ( occupancy, 1 )
        else:
            perf.observe ( occupancy, 0 )


labels = { False: '--', True: 'Used' }
print ( '{i0:10.8s}__{i1:10.8s}__{i2:10.8s}__{i3:10.8s}__{i4:10.8s}_______{f1:6.3f}'.format (
    i0 = labels[use_co2],
    i1 = labels[use_humidity],
    i2 = labels[use_humratio],
    i3 = labels[use_light],
    i4 = labels[use_temperature],

    f1 = perf.get_f1(),
) )
```

```
print ( '' )
print ( 'Analysing:', DATA_FILE )
print ( '' )
print ( 'CO2_____Humidity____Hum._Ratio__Light_____Temperature_____f1' )

for i0 in [False, True]:
    for i1 in [False, True]:
        for i2 in [False, True]:
            for i3 in [False, True]:
                for i4 in [False, True]:
                    run_test ( i0, i1, i2, i3, i4 )
```

B.2.3 Analyse model C

```
import csv
import time
import sys

from SubjectiveLogic import sl
from Tools import sensor
from Tools import pm

#DATA_FILE = 'data\\occupancy\\test.csv'
DATA_FILE = 'data\\occupancy\\train.csv'
#DATA_FILE = 'data\\occupancy\\validate.csv'


def run_test(use_co2, use_humidity, use_humratio, use_light, use_temperature):
    cond_list = { False:
                    { False:
                        { False:
                            { False: { False: 0.00, True: 0.48 }, True: { False: 0.90, True: 0.77 } },
                          True:
                            { False: { False: 0.49, True: 0.42 }, True: { False: 0.56, True: 0.74 } }
                        },
                      True:
                        { False:
                            { False: { False: 0.39, True: 0.41 }, True: { False: 0.57, True: 0.56 } },
                          True:
                            { False: { False: 0.42, True: 0.45 }, True: { False: 0.56, True: 0.54 } }
                        }
                    },
                  True:
                    { False:
                        { False:
                            { False: { False: 0.30, True: 0.46 }, True: { False: 0.41, True: 0.55 } },
                          True:
                            { False: { False: 0.33, True: 0.45 }, True: { False: 0.36, True: 0.51 } }
                        },
                      True:
                        { False:
                            { False: { False: 0.39, True: 0.42 }, True: { False: 0.55, True: 0.51 } },
                          True:
                            { False: { False: 0.41, True: 0.40 }, True: { False: 0.53, True: 0.53 } }
                        }
                    }
                }
    cond = cond_list[use_co2][use_humidity][use_humratio][use_light][use_temperature]

    evidence_co2         = sl.Evidence ( gamma = 0.90, delta = 0.90 )
    evidence_humidity    = sl.Evidence ( gamma = 0.40, delta = 1.00 )
    evidence_humratio    = sl.Evidence ( gamma = 0.91, delta = 0.93 )
    evidence_light       = sl.Evidence ( gamma = 0.24, delta = 0.20 )
    evidence_temperature = sl.Evidence ( gamma = 0.90, delta = 0.90 )

    sensor_co2         = sensor.Sensor ( lower_bound =  410.0      , upper_bound = 1210.0      , rate =    3.000    )
    sensor_humidity    = sensor.Sensor ( lower_bound =   38.7      , upper_bound =   40.7      , rate =    0.160    )
    sensor_humratio    = sensor.Sensor ( lower_bound =    0.004270, upper_bound =    0.006070, rate =    0.000001 )
    sensor_light       = sensor.Sensor ( lower_bound =  200.0      , upper_bound =  400.0      , rate =   63.000    )
    sensor_temperature = sensor.Sensor ( lower_bound =   19.6      , upper_bound =   23.2      , rate =    0.008    )

    base_co2         = { 'empty': 0.10, 'occupied': 0.90 }
    base_humidity    = { 'empty': 0.01, 'occupied': 0.99 }
    base_humratio    = { 'empty': 0.50, 'occupied': 0.50 }
    base_light       = { 'empty': 0.10, 'occupied': 0.90 }
    base_temperature = { 'empty': 0.15, 'occupied': 0.85 }

    domain = [ 'empty', 'occupied' ]
    perf = pm.PerfMeasure()

    with open ( DATA_FILE ) as csvfile:
        csvreader = csv.DictReader ( csvfile, delimiter = ',', quotechar = '"', skipinitialspace = True )
        for row in csvreader:
            evidence_co2.observe         ( sensor_co2.convert         ( float ( row['CO2']           ) ) )
            evidence_humidity.observe    ( sensor_humidity.convert    ( float ( row['Humidity']      ) ) )
            evidence_humratio.observe    ( sensor_humratio.convert    ( float ( row['HumidityRatio'] ) ) )
            evidence_light.observe       ( sensor_light.convert       ( float ( row['Light']         ) ) )
            evidence_temperature.observe ( sensor_temperature.convert ( float ( row['Temperature']   ) ) )

            fused = sl.Opinion (
                domain = domain,
                belief = { 'empty': 0, 'occupied': 0 },
                base = { 'empty': 0.50, 'occupied': 0.50 },
            )
```

```
            if use_co2:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_co2.get_disbelief(),
                                   'occupied': evidence_co2.get_belief() },
                        base = base_co2,
                    )
                )

            if use_humidity:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_humidity.get_disbelief(),
                                   'occupied': evidence_humidity.get_belief() },
                        base = base_humidity,
                    )
                )

            if use_humratio:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_humratio.get_disbelief(),
                                   'occupied': evidence_humratio.get_belief() },
                        base = base_humratio,
                    )
                )

            if use_light:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_light.get_disbelief(),
                                   'occupied': evidence_light.get_belief() },
                        base = base_light,
                    )
                )

            if use_temperature:
                fused = sl.cumulative_fusion (
                    fused,
                    sl.Opinion (
                        domain = domain,
                        belief = { 'empty': evidence_temperature.get_disbelief(),
                                   'occupied': evidence_temperature.get_belief() },
                        base = base_temperature,
                    )
                )

            occupancy = int(row['Occupancy'])

            if fused.belief['occupied'] + fused.base['occupied'] * fused.uncert > cond:
                perf.observe ( occupancy, 1 )
            else:
                perf.observe ( occupancy, 0 )


    labels = { False: '--', True: 'Used' }
    print ( '{i0:10.8s}__{i1:10.8s}__{i2:10.8s}__{i3:10.8s}__{i4:10.8s}_______{f1:6.3f}'.format (
        i0 = labels[use_co2],
        i1 = labels[use_humidity],
        i2 = labels[use_humratio],
        i3 = labels[use_light],
        i4 = labels[use_temperature],

        f1 = perf.get_f1(),
    ) )


print ( '' )
print ( 'Analysing:', DATA_FILE )
print ( '' )
print ( 'CO2__________Humidity____Hum._Ratio__Light________Temperature_____f1' )

for i0 in [False, True]:
    for i1 in [False, True]:
        for i2 in [False, True]:
            for i3 in [False, True]:
                for i4 in [False, True]:
                    run_test ( i0, i1, i2, i3, i4 )
```

B.2.4 Optimise parameters, model A

```
import csv
import math
import sys

from SubjectiveLogic import sl
```

```
from Tools import sensor
from Tools import pm


sensor_name = None
sensor_values = []
occupancy_values = []

GREEK_SCALE = 0.01      # Greek-letter parameters: gamma and delta
bound_scale = None
rate_scale = None

value_low = None
value_high = None
rate_low = None
rate_high = None

GREEK_STEP = 1
BOUND_STEP = 1
RATE_STEP = 1

best_f1 = -1
best_gamma = None
best_delta = None
best_lower_bound = None
best_upper_bound = None
best_rate = None


def analyse():
    global sensor_values
    global occupancy_values

    lowest_value = 2000000000
    highest_value = -2000000000

    lowest_diff = 2000000000
    highest_diff = -2000000000

    value = 0
    diff = None

    with open ( 'data\\occupancy\\train.csv' ) as csvfile:
        csvreader = csv.DictReader ( csvfile, delimiter = ',', quotechar = '"', skipinitialspace = True )
        for row in csvreader:
            occupancy = int ( row['Occupancy'] )

            last_value = value
            value = float ( row[sensor_name] )

            lowest_value = min ( value, lowest_value )
            highest_value = max ( value, highest_value )

            if diff == None:
                diff = 0
            else:
                diff = abs ( value - last_value )

                lowest_diff = min ( diff, lowest_diff )
                highest_diff = max ( diff, highest_diff )

            sensor_values.append ( value )
            occupancy_values.append ( occupancy )

        print ( '' )
        print ( 'Sensor analysed: {sensor}\n'.format ( sensor = sensor_name ) )
        print ( 'Values:\nLowest = {lowest}\nHighest = {highest}\n'.format (
            lowest = lowest_value, highest = highest_value
        ) )
        print ( 'Diffs:\nLowest = {lowest}\nHighest = {highest}\n'.format (
            lowest = lowest_diff, highest = highest_diff
        ) )

        return ( lowest_value, highest_value, lowest_diff, highest_diff )


def run_test(gamma, delta, lower_bound, upper_bound, rate):
    evidence = sl.Evidence ( gamma = gamma * GREEK_SCALE, delta = delta * GREEK_SCALE )
    sensor_model = sensor.Sensor ( lower_bound = lower_bound * bound_scale,
                                   upper_bound = upper_bound * bound_scale,
                                   rate = rate * rate_scale )
    perf = pm.PerfMeasure()

    for i in range ( len ( sensor_values ) ):
        occupancy = occupancy_values[i]
        value = sensor_values[i]

        sensor_result = sensor_model.convert ( value )
        evidence.observe ( sensor_result )

        if evidence.get_belief() > evidence.get_disbelief():
            perf.observe ( occupancy, 1 )
        else:
            perf.observe ( occupancy, 0 )

    f1 = perf.get_f1()

    update_best ( gamma, delta, lower_bound, upper_bound, rate, f1 )
```

```
def update_best(gamma, delta, lower_bound, upper_bound, rate, f1):
    global best_f1
    global best_gamma
    global best_delta
    global best_lower_bound
    global best_upper_bound
    global best_rate

    if f1 > best_f1:
        best_f1 = f1
        best_gamma = gamma
        best_delta = delta
        best_lower_bound = lower_bound
        best_upper_bound = upper_bound
        best_rate = rate


def show_params(gamma, delta, lower_bound, upper_bound, rate, f1):
    print ( '{gamma:0.2f}____{delta:0.2f}____{lower_bound:8.6f}____{upper_bound:8.6f}____{rate:8.6f}'
            '____{f1:8.6f}'.format (
        gamma = gamma * GREEK_SCALE,
        delta = delta * GREEK_SCALE,
        lower_bound = lower_bound * bound_scale,
        upper_bound = upper_bound * bound_scale,
        rate = rate * rate_scale,
        f1 = f1,
    ) )


def optimise():
    print ( 'Used ranges:' )
    print ( 'Greek: {low:4d} {step:4d} {high:4d}'.format ( low = 0, high = 100, step = GREEK_STEP ) )
    print ( 'Value: {low:4d} {step:4d} {high:4d}'.format ( low = value_low, high = value_high, step = BOUND_STEP ) )
    print ( 'Rate:  {low:4d} {step:4d} {high:4d}'.format ( low = rate_low, high = rate_high, step = RATE_STEP ) )
    print ( '' )

    for gamma in range ( 0, 101, GREEK_STEP ):
        for delta in range ( 0, 101, GREEK_STEP ):
            for lower_bound in range ( value_low, value_high, BOUND_STEP ):
                for upper_bound in range ( lower_bound + BOUND_STEP, value_high + BOUND_STEP, BOUND_STEP ):
                    for rate in range ( rate_low, rate_high + RATE_STEP, RATE_STEP ):
                        run_test ( gamma, delta, lower_bound, upper_bound, rate )


# Main

sensor_name = sys.argv[1]

( lowest_value, highest_value, lowest_diff, highest_diff ) = analyse()


# Normalise magnitude of data by scaling
value_range = highest_value - lowest_value
value_range_scale = math.ceil ( math.log ( value_range, 10 ) )
bound_scale = 10 ** ( value_range_scale ) * 0.001

rate_range = highest_diff - lowest_diff
rate_range_scale = math.ceil ( math.log ( rate_range, 10 ) )
rate_scale = 10 ** ( rate_range_scale ) * 0.001

value_low = math.floor ( lowest_value / bound_scale )
value_high = math.ceil ( highest_value / bound_scale )
rate_low = math.floor ( lowest_diff / rate_scale )
rate_high = math.ceil ( highest_diff / rate_scale )


optimise()

print ( '' )
print ( 'Gamma   Delta   Lower Bound  Upper Bound  Rate of C   F1' )
show_params ( best_gamma, best_delta, best_lower_bound, best_upper_bound, best_rate, best_f1 )
print ( '' )
```

B.2.5 Optimise parameters, model C

```
import csv
import math
import sys

from SubjectiveLogic import sl
from Tools import sensor
from Tools import pm


MAX_OPT_ITERS = 10

sensor_name = None
sensor_values = []
occupancy_values = []

GREEK_SCALE = 0.01      # Greek-letter parameters: gamma and delta
bound_scale = None
rate_scale = None
BASE_SCALE = 0.01
```

```
THRES_SCALE = 0.01

value_low = None
value_high = None
rate_low = None
rate_high = None

GREEK_STEP_QUICK = 10
BOUND_STEP_QUICK = 20
RATE_STEP_QUICK = 20
BASE_STEP_QUICK = 20
THRES_STEP_QUICK = 20

GREEK_STEP_DETAIL = 1
BOUND_STEP_DETAIL = 1
RATE_STEP_DETAIL = 1
BASE_STEP_DETAIL = 1
THRES_STEP_DETAIL = 1

best_f1 = -1
best_gamma = None
best_delta = None
best_lower = None
best_upper = None
best_rate = None
best_base = None
best_thres = None


def analyse():
    global sensor_values
    global occupancy_values

    lowest_value = 2000000000
    highest_value = -2000000000

    lowest_diff = 2000000000
    highest_diff = -2000000000

    value = 0
    diff = None

    with open ( 'data\\occupancy\\train.csv' ) as csvfile:
        csvreader = csv.DictReader ( csvfile, delimiter = ',', quotechar = '"', skipinitialspace = True )
        for row in csvreader:
            occupancy = int ( row['Occupancy'] )

            last_value = value
            value = float ( row[sensor_name] )

            lowest_value = min ( value, lowest_value )
            highest_value = max ( value, highest_value )

            if diff == None:
                diff = 0
            else:
                diff = abs ( value - last_value )

                lowest_diff = min ( diff, lowest_diff )
                highest_diff = max ( diff, highest_diff )

            sensor_values.append ( value )
            occupancy_values.append ( occupancy )

        print ( '' )
        print ( 'Sensor analysed: {sensor}\n'.format ( sensor = sensor_name ) )
        print ( 'Values:\nLowest = {lowest}\nHighest = {highest}\n'.format (
            lowest = lowest_value, highest = highest_value
        ) )
        print ( 'Diffs:\nLowest = {lowest}\nHighest = {highest}\n'.format (
            lowest = lowest_diff, highest = highest_diff
        ) )

        return ( lowest_value, highest_value, lowest_diff, highest_diff )


def run_test(gamma, delta, lower, upper, rate, base, thres, display_update = True):
    evidence = sl.Evidence ( gamma = gamma * GREEK_SCALE, delta = delta * GREEK_SCALE )
    sensor_model = sensor.Sensor ( lower = lower * bound_scale,
                                   upper = upper * bound_scale,
                                   rate = rate * rate_scale )
    perf = pm.PerfMeasure()

    for i in range ( len ( sensor_values ) ):
        occupancy = occupancy_values[i]
        value = sensor_values[i]

        sensor_result = sensor_model.convert ( value )
        evidence.observe ( sensor_result )

        if evidence.get_belief() + evidence.get_uncert() * ( base * BASE_SCALE ) > ( thres * THRES_SCALE ):
            perf.observe ( occupancy, 1 )
        else:
            perf.observe ( occupancy, 0 )

    f1 = perf.get_f1()

    if display_update and ( f1 > best_f1 ):
        show_params ( gamma, delta, lower, upper, rate, base, thres, f1 )
```

```
    update_best ( gamma, delta, lower, upper, rate, base, thres, f1 )


def update_best(gamma, delta, lower, upper, rate, base, thres, f1):
    global best_f1
    global best_gamma
    global best_delta
    global best_lower
    global best_upper
    global best_rate
    global best_base
    global best_thres

    if f1 > best_f1:
        best_f1 = f1
        best_gamma = gamma
        best_delta = delta
        best_lower = lower
        best_upper = upper
        best_rate = rate
        best_base = base
        best_thres = thres


def show_params(gamma, delta, lower, upper, rate, base, thres, f1):
    print ( '{gamma:0.2f}    {delta:0.2f}    {lower:8.6f}    {upper:8.6f}    {rate:8.6f}    {base:0.2f}'
            '    {thres:0.2f}    {f1:8.6f}'.format (
        gamma = gamma * GREEK_SCALE,
        delta = delta * GREEK_SCALE,
        lower = lower * bound_scale,
        upper = upper * bound_scale,
        rate = rate * rate_scale,
        base = base * BASE_SCALE,
        thres = thres * THRES_SCALE,
        f1 = f1,
    ) )


def optimise_all():
    print ( 'Used ranges:' )
    print ( 'Greek: {low:4d} {step:4d} {high:4d}'.format (
        low = 0, high = 100, step = GREEK_STEP_QUICK
    ) )
    print ( 'Value: {low:4d} {step:4d} {high:4d}'.format (
        low = value_low, high = value_high, step = BOUND_STEP_QUICK
    ) )
    print ( 'Rate:  {low:4d} {step:4d} {high:4d}  (full range)'.format (
        low = rate_low, high = rate_high, step = RATE_STEP_QUICK
    ) )
    print ( 'Rate:  {low:4d} {step:4d} {high:4d}  (likely range)'.format (
        low = rate_low, high = rate_low + 10 * RATE_STEP_DETAIL, step = RATE_STEP_DETAIL
    ) )
    print ( 'Base:  {low:4d} {step:4d} {high:4d}'.format (
        low = 10, high = 90, step = BASE_STEP_QUICK
    ) )
    print ( 'Thres: {low:4d} {step:4d} {high:4d}'.format (
        low = 10, high = 90, step = THRES_STEP_QUICK
    ) )
    print ( '' )

    for gamma in range ( 0, 101, GREEK_STEP_QUICK ):
        for delta in range ( 0, 101, GREEK_STEP_QUICK ):
            for lower in range ( value_low, value_high, BOUND_STEP_QUICK ):
                for upper in range ( lower + BOUND_STEP_QUICK, value_high + BOUND_STEP_QUICK, BOUND_STEP_QUICK ):
                    # Try full range of possible rate-of-changes
                    for rate in range ( rate_low, rate_high + RATE_STEP_QUICK, RATE_STEP_QUICK ):
                        for base in range ( 10, 91, BASE_STEP_QUICK ):
                            for thres in range ( 10, 91, THRES_STEP_QUICK ):
                                run_test ( gamma, delta, lower, upper, rate, base, thres, False )

                    # Pay extra attention to small values
                    for rate in range ( rate_low, rate_low + 11 * RATE_STEP_DETAIL, RATE_STEP_DETAIL ):
                        for base in range ( 10, 91, BASE_STEP_QUICK ):
                            for thres in range ( 10, 91, THRES_STEP_QUICK ):
                                run_test ( gamma, delta, lower, upper, rate, base, thres, False )


def optimise_gamma():
    print ( '' )
    print ( 'Testing gamma      from {low:4.2f} to {high:4.2f}    incrementing by {incr:4.2f}'.format (
        low = 0 * GREEK_SCALE, high = 100 * GREEK_SCALE, incr = GREEK_STEP_DETAIL * GREEK_SCALE
    ) )
    for gamma in range ( 0, 101, GREEK_STEP_DETAIL ):
        run_test ( gamma, best_delta, best_lower, best_upper, best_rate, best_base, best_thres )


def optimise_delta():
    print ( '' )
    print ( 'Testing delta      from {low:4.2f} to {high:4.2f}    incrementing by {incr:4.2f}'.format (
        low = 0 * GREEK_SCALE, high = 100 * GREEK_SCALE, incr = GREEK_STEP_DETAIL * GREEK_SCALE
    ) )
    for delta in range ( 0, 101, GREEK_STEP_DETAIL ):
        run_test ( best_gamma, delta, best_lower, best_upper, best_rate, best_base, best_thres )


def optimise_bounds():
    print ( '' )
    print ( 'Testing bounds     from {low:10.6f} to {high:10.6f}        incrementing by {incr:10.6f}'.format (
```

```
        low = value_low * bound_scale, high = value_high * bound_scale, incr = BOUND_STEP_DETAIL * bound_scale
    ) )
    for lower in range ( value_low, value_high, BOUND_STEP_QUICK ):
        for upper in range ( lower + BOUND_STEP_QUICK, value_high + BOUND_STEP_QUICK, BOUND_STEP_QUICK ):
            run_test ( best_gamma, best_delta, lower, upper, best_rate, best_base, best_thres )


def optimise_rate():
    print ( '' )
    print ( 'Testing rate       from {low:10.6f} to {high:10.6f}         incrementing by {incr:10.6f}'.format (
        low = rate_low * rate_scale, high = rate_high * rate_scale, incr = RATE_STEP_DETAIL * rate_scale
    ) )
    for rate in range ( rate_low, rate_high + RATE_STEP_DETAIL, RATE_STEP_DETAIL ):
        run_test ( best_gamma, best_delta, best_lower, best_upper, rate, best_base, best_thres )


def optimise_base():
    print ( '' )
    print ( 'Testing base rate  from {low:4.2f} to {high:4.2f}    incrementing by {incr:4.2f}'.format (
        low = 0 * BASE_SCALE, high = 100 * BASE_SCALE, incr = BASE_STEP_DETAIL * BASE_SCALE
    ) )
    for base in range ( 0, 101, BASE_STEP_DETAIL ):
        run_test ( best_gamma, best_delta, best_lower, best_upper, best_rate, base, best_thres )


def optimise_thres():
    print ( '' )
    print ( 'Testing threshold  from {low:4.2f} to {high:4.2f}    incrementing by {incr:4.2f}'.format (
        low = 0 * THRES_SCALE, high = 100 * THRES_SCALE, incr = THRES_STEP_DETAIL * THRES_SCALE
    ) )
    for thres in range ( 0, 101, THRES_STEP_DETAIL ):
        run_test ( best_gamma, best_delta, best_lower, best_upper, best_rate, best_base, thres )



# Main

if len ( sys.argv ) == 1:
    sensor_name = 'CO2'
else:
    sensor_name = sys.argv[1]


( lowest_value, highest_value, lowest_diff, highest_diff ) = analyse()


# Normalise magnitude of data by scaling
value_range = highest_value - lowest_value
value_range_scale = math.ceil ( math.log ( value_range, 10 ) )
bound_scale = 10 ** ( value_range_scale ) * 0.001

rate_range = highest_diff - lowest_diff
rate_range_scale = math.ceil ( math.log ( rate_range, 10 ) )
rate_scale = 10 ** ( rate_range_scale ) * 0.001

value_low = math.floor ( lowest_value / bound_scale )
value_high = math.ceil ( highest_value / bound_scale )
rate_low = math.floor ( lowest_diff / rate_scale )
rate_high = math.ceil ( highest_diff / rate_scale )

print ( '' )
print ( 'Running coarse optimisation...' )
print ( '' )

optimise_all()

print ( '' )
print ( 'Best result after coarse optimisation:' )
print ( 'Gamma   Delta   Lower Bound  Upper Bound  Rate-of-C   Base Rate   Threshold   F1' )
show_params ( best_gamma, best_delta, best_lower, best_upper, best_rate, best_base, best_thres, best_f1 )
print ( '' )


improvement = True
i = 0

print ( '' )
print ( 'Running fine optimisation...' )

while improvement and ( i < MAX_OPT_ITERS ):
    print ( '' )
    print ( 'Running iteration', i )
    old_f1 = best_f1

    optimise_gamma()
    optimise_delta()
    optimise_bounds()
    optimise_rate()
    optimise_base()
    optimise_thres()

    if best_f1 > old_f1:
        improvement = True
    else:
        improvement = False
        print ( 'No further improvement found; cancelling further iterations.' )
    i = i + 1

print ( '' )
```

```
print ( 'Best result after fine optimisation:' )
show_params ( best_gamma, best_delta, best_lower, best_upper, best_rate, best_base, best_thres, best_f1 )
print ( '' )
```

B.3 Location detection

B.3.1 Analyse

```
import math
import sys
import datetime

from Tools import delayed_csv
from Tools import interpolated_csv


DATA_PATH = '..\\data\\Run_2\\'

WALL_OFFSET = -0.8
WINDOW_OFFSET = -8.0

GRID_SIZE = 100      ## mm
X_MIN = 2050
X_MAX = 5550
Y_MIN =  650
Y_MAX = 3750

SIGMA = 250
PREV_WEIGHT = 0.7

WALL_X = 3400
WALL_Y = 4000
WINDOW_X = 1750
WINDOW_Y = 2000


# Pseudo-constant; initialised by code
DOMAIN = None


def coord_to_name(x, y):
    return 'x{x:04d}y{y:04d}'.format ( x = x, y = y )


def name_to_coord(name):
    x = int ( name[1:5] )
    y = int ( name[6:10] )

    return ( x, y )


def _get_domain():
    domain = []

    for x in range ( X_MIN, X_MAX, GRID_SIZE ):
        for y in range ( Y_MIN, Y_MAX, GRID_SIZE ):
            domain.append ( coord_to_name ( x, y ) )

    return domain


def pyth(x0, y0, x1, y1):
    return math.sqrt ( ( x0 - x1 ) ** 2 + ( y0 - y1 ) ** 2 )


def empty_map():
    result = {}

    for cell in DOMAIN:
        result[cell] = 0

    return result


def add_maps(a, b):
    result = {}

    for cell in DOMAIN:
        result[cell] = a[cell] + b[cell]

    return result


def scale_map(density, factor):
    result = {}

    for cell in DOMAIN:
        result[cell] = density[cell] * factor

    return result


def find_max(density):
    max_density = -1
```

```
    max_cell = None

    for cell in DOMAIN:
        if density[cell] > max_density:
            max_density = density[cell]
            max_cell = cell

    if density[max_cell] == 0:
        print ( 'Warning: find_max() called on empty map' )

    coord = name_to_coord ( max_cell )
    result = { 'x': coord[0] + GRID_SIZE / 2, 'y': coord[1] + GRID_SIZE / 2 }

    return result


def calc_all_probabilities(x_mu, y_mu, sigma):
    result = {}

    for cell in DOMAIN:
        result[cell] = calc_one_probability(cell, x_mu, y_mu, sigma)

    return result


def calc_one_probability(cell, x_mu, y_mu, sigma):
    ( x_min, y_min ) = name_to_coord ( cell )
    x_max = x_min + GRID_SIZE
    y_max = y_min + GRID_SIZE

    denom = math.sqrt ( 2 ) * sigma
    x0 = ( x_min - x_mu ) / denom
    x1 = ( x_max - x_mu ) / denom
    y0 = ( y_min - y_mu ) / denom
    y1 = ( y_max - y_mu ) / denom

    ## Reduce number of calls to math.erf() in order to reduce calculation time
    ## (this does not affect accuracy in any meaningful way).

    if x0 < -3:
        erf_x0 = -1
    elif 3 < x0:
        erf_x0 = 1
    else: # -3 <= x0 <= 3
        erf_x0 = math.erf ( x0 )

    if x1 < -3:
        erf_x1 = -1
    elif 3 < x1:
        erf_x1 = 1
    else: # -3 <= x1 <= 3
        erf_x1 = math.erf ( x1 )

    if y0 < -3:
        erf_y0 = -1
    elif 3 < y0:
        erf_y0 = 1
    else: # -3 <= y0 <= 3
        erf_y0 = math.erf ( y0 )

    if y1 < -3:
        erf_y1 = -1
    elif 3 < y1:
        erf_y1 = 1
    else: # -3 <= y1 <= 3
        erf_y1 = math.erf ( y1 )

    return 0.25 * ( erf_x1 - erf_x0 ) * ( erf_y1 - erf_y0 )


def estimate_location(detections, previous_map):
    result = scale_map ( previous_map, PREV_WEIGHT )

    for detection in detections:
        density = calc_all_probabilities ( detection['x'], detection['y'], SIGMA )
        result = add_maps ( result, density )

    return result


def analyse(experiment):
    analyse_start = datetime.datetime.now()
    print ( '' )
    print ( 'Doing init...' )

    base_filename = DATA_PATH + 'experiment{id:02d}_'.format ( id = experiment )

    csv_gt = interpolated_csv.InterpolatedCsv ( base_filename + 'ground_truth.txt' )
    csv_wall = delayed_csv.DelayedCsv ( base_filename + 'wall.txt', WALL_OFFSET )
    csv_window = delayed_csv.DelayedCsv ( base_filename + 'window.txt', WINDOW_OFFSET )

    time_start = max ( csv_gt.time_start, csv_wall.time_start, csv_window.time_start )
    time_stop = min ( csv_gt.time_stop, csv_wall.time_stop, csv_window.time_stop )

    sl_wall_map = empty_map()
    for time in range ( math.ceil ( csv_wall.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_wall_map = estimate_location ( csv_wall.get_detections ( time / 100 ), sl_wall_map )

    sl_window_map = empty_map()
```

```
    for time in range ( math.ceil ( csv_window.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_window_map = estimate_location ( csv_window.get_detections ( time / 100 ), sl_window_map )

    prop_wall_diff = 0
    prop_window_diff = 0
    prop_fused1_diff = 0
    prop_fused2a_diff = 0
    prop_fused2b_diff = 0
    prop_fused2c_diff = 0
    prop_fused2d_diff = 0
    sl_wall_diff = 0
    sl_window_diff = 0
    sl_fused1_diff = 0
    sl_fused2a_diff = 0
    sl_fused2b_diff = 0
    sl_fused2c_diff = 0
    sl_fused2d_diff = 0
    sl_fused2e_diff = 0
    sl_fused2f_diff = 0
    sl_fused2g_diff = 0
    samples = 0

    print ( 'Doing loop...' )
    for time in range ( math.ceil ( time_start * 100), math.floor ( time_stop * 100 ), 25 ):
        # Ground truth
        gt = csv_gt.get ( time / 100 )

        # Location as determined by proprietary algorithm
        prop_wall_loc = csv_wall.get_location ( time / 100, False )
        prop_window_loc = csv_window.get_location ( time / 100, False )

        # Sensor fusion using results from proprietary algorithm (straight average)
        prop_fused1_loc = {
            'x': ( prop_wall_loc['x'] + prop_window_loc['x'] ) / 2,
            'y': ( prop_wall_loc['y'] + prop_window_loc['y'] ) / 2
        }

        # Sensor fusion using results from proprietary algorithm (weighted average)
        prop_wall_dist = pyth ( WALL_X, WALL_Y, prop_wall_loc['x'], prop_wall_loc['y'] )
        prop_window_dist = pyth ( WINDOW_X, WINDOW_Y, prop_window_loc['x'], prop_window_loc['y'] )

        if prop_wall_dist < prop_window_dist:
            prop_fused2a_wall_weight = 1
            prop_fused2b_wall_weight = 1
            prop_fused2c_wall_weight = 1
            prop_fused2d_wall_weight = 1
            prop_fused2a_window_weight = ( prop_wall_dist / prop_window_dist ) ** 1
            prop_fused2b_window_weight = ( prop_wall_dist / prop_window_dist ) ** 2
            prop_fused2c_window_weight = ( prop_wall_dist / prop_window_dist ) ** 4
            prop_fused2d_window_weight = ( prop_wall_dist / prop_window_dist ) ** 8
        else:
            prop_fused2a_wall_weight = ( prop_window_dist / prop_wall_dist ) ** 1
            prop_fused2b_wall_weight = ( prop_window_dist / prop_wall_dist ) ** 2
            prop_fused2c_wall_weight = ( prop_window_dist / prop_wall_dist ) ** 4
            prop_fused2d_wall_weight = ( prop_window_dist / prop_wall_dist ) ** 8
            prop_fused2a_window_weight = 1
            prop_fused2b_window_weight = 1
            prop_fused2c_window_weight = 1
            prop_fused2d_window_weight = 1


        prop_fused2a_loc = {
            'x': ( prop_wall_loc['x'] * prop_fused2a_wall_weight +
                   prop_window_loc['x'] * prop_fused2a_window_weight ) /
                 ( prop_fused2a_wall_weight + prop_fused2a_window_weight ),
            'y': ( prop_wall_loc['y'] * prop_fused2a_wall_weight +
                   prop_window_loc['y'] * prop_fused2a_window_weight ) /
                 ( prop_fused2a_wall_weight + prop_fused2a_window_weight )
        }
        prop_fused2b_loc = {
            'x': ( prop_wall_loc['x'] * prop_fused2b_wall_weight +
                   prop_window_loc['x'] * prop_fused2b_window_weight ) /
                 ( prop_fused2b_wall_weight + prop_fused2b_window_weight ),
            'y': ( prop_wall_loc['y'] * prop_fused2b_wall_weight +
                   prop_window_loc['y'] * prop_fused2b_window_weight ) /
                 ( prop_fused2b_wall_weight + prop_fused2b_window_weight )
        }
        prop_fused2c_loc = {
            'x': ( prop_wall_loc['x'] * prop_fused2c_wall_weight +
                   prop_window_loc['x'] * prop_fused2c_window_weight ) /
                 ( prop_fused2c_wall_weight + prop_fused2c_window_weight ),
            'y': ( prop_wall_loc['y'] * prop_fused2c_wall_weight +
                   prop_window_loc['y'] * prop_fused2c_window_weight ) /
                 ( prop_fused2c_wall_weight + prop_fused2c_window_weight )
        }
        prop_fused2d_loc = {
            'x': ( prop_wall_loc['x'] * prop_fused2d_wall_weight +
                   prop_window_loc['x'] * prop_fused2d_window_weight ) /
                 ( prop_fused2c_wall_weight + prop_fused2d_window_weight ),
            'y': ( prop_wall_loc['y'] * prop_fused2d_wall_weight +
                   prop_window_loc['y'] * prop_fused2d_window_weight ) /
                 ( prop_fused2c_wall_weight + prop_fused2d_window_weight )
        }

        # Calculate error
        prop_wall_diff    = prop_wall_diff +
                            pyth ( gt['x'], gt['y'], prop_wall_loc['x'], prop_wall_loc['y'] ) ** 2
        prop_window_diff  = prop_window_diff +
                            pyth ( gt['x'], gt['y'], prop_window_loc['x'], prop_window_loc['y'] ) ** 2
```

```
        prop_fused1_diff  = prop_fused1_diff +
                            pyth ( gt['x'], gt['y'], prop_fused1_loc['x'], prop_fused1_loc['y'] ) ** 2
        prop_fused2a_diff = prop_fused2a_diff +
                            pyth ( gt['x'], gt['y'], prop_fused2a_loc['x'], prop_fused2a_loc['y'] ) ** 2
        prop_fused2b_diff = prop_fused2b_diff +
                            pyth ( gt['x'], gt['y'], prop_fused2b_loc['x'], prop_fused2b_loc['y'] ) ** 2
        prop_fused2c_diff = prop_fused2c_diff +
                            pyth ( gt['x'], gt['y'], prop_fused2c_loc['x'], prop_fused2c_loc['y'] ) ** 2
        prop_fused2d_diff = prop_fused2d_diff +
                            pyth ( gt['x'], gt['y'], prop_fused2d_loc['x'], prop_fused2d_loc['y'] ) ** 2


        # Location as determined by subjective logic algorithm
        wall_raw = csv_wall.get_detections ( time / 100 )
        window_raw = csv_window.get_detections ( time / 100 )

        sl_wall_map = estimate_location ( wall_raw, sl_wall_map )
        sl_window_map = estimate_location ( window_raw, sl_window_map )

        sl_wall_loc = find_max ( sl_wall_map )
        sl_window_loc = find_max ( sl_window_map )

        # SL sensor fusion (basic approach)
        sl_fused1_map = add_maps ( sl_wall_map, sl_window_map )

        sl_fused1_loc = find_max ( sl_fused1_map )

        # SL sensor fusion (improved approach)
        sl_wall_dist = pyth ( WALL_X, WALL_Y, sl_wall_loc['x'], sl_wall_loc['y'] )
        sl_window_dist = pyth ( WINDOW_X, WINDOW_Y, sl_window_loc['x'], sl_window_loc['y'] )

        if sl_wall_dist < sl_window_dist:
            sl_window_map_weighted_a = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 1 )
            sl_window_map_weighted_b = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 2 )
            sl_window_map_weighted_c = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 4 )
            sl_window_map_weighted_d = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 8 )
            sl_window_map_weighted_e = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 16 )
            sl_window_map_weighted_f = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 32 )
            sl_window_map_weighted_g = scale_map ( sl_window_map, ( sl_wall_dist / sl_window_dist ) ** 64 )
            sl_fused2a_map = add_maps ( sl_wall_map, sl_window_map_weighted_a )
            sl_fused2b_map = add_maps ( sl_wall_map, sl_window_map_weighted_b )
            sl_fused2c_map = add_maps ( sl_wall_map, sl_window_map_weighted_c )
            sl_fused2d_map = add_maps ( sl_wall_map, sl_window_map_weighted_d )
            sl_fused2e_map = add_maps ( sl_wall_map, sl_window_map_weighted_e )
            sl_fused2f_map = add_maps ( sl_wall_map, sl_window_map_weighted_f )
            sl_fused2g_map = add_maps ( sl_wall_map, sl_window_map_weighted_g )
        else:
            sl_wall_map_weighted_a = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 1 )
            sl_wall_map_weighted_b = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 2 )
            sl_wall_map_weighted_c = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 4 )
            sl_wall_map_weighted_d = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 8 )
            sl_wall_map_weighted_e = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 16 )
            sl_wall_map_weighted_f = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 32 )
            sl_wall_map_weighted_g = scale_map ( sl_wall_map, ( sl_window_dist / sl_wall_dist ) ** 64 )
            sl_fused2a_map = add_maps ( sl_wall_map_weighted_a, sl_window_map )
            sl_fused2b_map = add_maps ( sl_wall_map_weighted_b, sl_window_map )
            sl_fused2c_map = add_maps ( sl_wall_map_weighted_c, sl_window_map )
            sl_fused2d_map = add_maps ( sl_wall_map_weighted_d, sl_window_map )
            sl_fused2e_map = add_maps ( sl_wall_map_weighted_e, sl_window_map )
            sl_fused2f_map = add_maps ( sl_wall_map_weighted_f, sl_window_map )
            sl_fused2g_map = add_maps ( sl_wall_map_weighted_g, sl_window_map )

        sl_fused2a_loc = find_max ( sl_fused2a_map )
        sl_fused2b_loc = find_max ( sl_fused2b_map )
        sl_fused2c_loc = find_max ( sl_fused2c_map )
        sl_fused2d_loc = find_max ( sl_fused2d_map )
        sl_fused2e_loc = find_max ( sl_fused2e_map )
        sl_fused2f_loc = find_max ( sl_fused2f_map )
        sl_fused2g_loc = find_max ( sl_fused2g_map )

        # Calculate error
        sl_wall_diff = sl_wall_diff + pyth ( gt['x'], gt['y'], sl_wall_loc['x'], sl_wall_loc['y'] ) ** 2
        sl_window_diff = sl_window_diff + pyth ( gt['x'], gt['y'], sl_window_loc['x'], sl_window_loc['y'] ) ** 2
        sl_fused1_diff = sl_fused1_diff + pyth ( gt['x'], gt['y'], sl_fused1_loc['x'], sl_fused1_loc['y'] ) ** 2
        sl_fused2a_diff = sl_fused2a_diff + pyth ( gt['x'], gt['y'], sl_fused2a_loc['x'], sl_fused2a_loc['y'] ) ** 2
        sl_fused2b_diff = sl_fused2b_diff + pyth ( gt['x'], gt['y'], sl_fused2b_loc['x'], sl_fused2b_loc['y'] ) ** 2
        sl_fused2c_diff = sl_fused2c_diff + pyth ( gt['x'], gt['y'], sl_fused2c_loc['x'], sl_fused2c_loc['y'] ) ** 2
        sl_fused2d_diff = sl_fused2d_diff + pyth ( gt['x'], gt['y'], sl_fused2d_loc['x'], sl_fused2d_loc['y'] ) ** 2
        sl_fused2e_diff = sl_fused2e_diff + pyth ( gt['x'], gt['y'], sl_fused2e_loc['x'], sl_fused2e_loc['y'] ) ** 2
        sl_fused2f_diff = sl_fused2f_diff + pyth ( gt['x'], gt['y'], sl_fused2f_loc['x'], sl_fused2f_loc['y'] ) ** 2
        sl_fused2g_diff = sl_fused2g_diff + pyth ( gt['x'], gt['y'], sl_fused2g_loc['x'], sl_fused2g_loc['y'] ) ** 2

        samples = samples + 1

    analyse_stop = datetime.datetime.now()

    print ( '' )
    print ( 'Results_____________RMS_error' )
    print ( '' )
    print ( 'Proprietary_algorithm:' )
    print ( '{name:20s}__{error:8.3f}'.format (
        name = 'Wall', error = math.sqrt ( prop_wall_diff / samples )
    ) )
    print ( '{name:20s}__{error:8.3f}'.format (
        name = 'Window', error = math.sqrt ( prop_window_diff / samples )
    ) )
    print ( '{name:20s}__{error:8.3f}'.format (
        name = 'Fused_(straight)', error = math.sqrt ( prop_fused1_diff / samples )
    ) )
```

```
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 1)', error = math.sqrt ( prop_fused2a_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 2)', error = math.sqrt ( prop_fused2b_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 4)', error = math.sqrt ( prop_fused2c_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 8)', error = math.sqrt ( prop_fused2d_diff / samples )
    ) )
    print ( '' )
    print ( 'Subjective logic algorithm:' )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Wall', error = math.sqrt ( sl_wall_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Window', error = math.sqrt ( sl_window_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (basic)', error = math.sqrt ( sl_fused1_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 1)', error = math.sqrt ( sl_fused2a_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 2)', error = math.sqrt ( sl_fused2b_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 4)', error = math.sqrt ( sl_fused2c_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 8)', error = math.sqrt ( sl_fused2d_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 16)', error = math.sqrt ( sl_fused2e_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 32)', error = math.sqrt ( sl_fused2f_diff / samples )
    ) )
    print ( '{name:20s} {error:8.3f}'.format (
        name = 'Fused (weighted, 64)', error = math.sqrt ( sl_fused2g_diff / samples )
    ) )
    print ( '' )
    print ( '' )
    print ( 'Calculation time:', int ( (analyse_stop - analyse_start).total_seconds() ), 'seconds' )

################################################################################

# Initialise pseudo-constant
DOMAIN = _get_domain()


if len ( sys.argv ) == 1:
    print ( '\nSpecify experiment number to analyse as first (and only) parameter.\n' )
    sys.exit()

experiment = int ( sys.argv[1] )
analyse ( experiment )
```

B.3.2 Optimise retention

```
import math
import sys
import datetime

from Tools import delayed_csv
from Tools import interpolated_csv


DATA_PATH = 'data\\location\\'

WALL_OFFSET = -0.8
WINDOW_OFFSET = -8.0

GRID_SIZE = 100      ## mm
X_MIN = 2050
X_MAX = 5550
Y_MIN =  650
Y_MAX = 3750

SIGMA = 250          ## Some initial value; will be optimised in next step

WALL_X = 3400
WALL_Y = 4000
WINDOW_X = 1750
WINDOW_Y = 2000


# Pseudo-constant; initialised by code
DOMAIN = None
```

```
def coord_to_name(x, y):
    return 'x{x:04d}y{y:04d}'.format ( x = x, y = y )


def name_to_coord(name):
    x = int ( name[1:5] )
    y = int ( name[6:10] )

    return ( x, y )


def _get_domain():
    domain = []

    for x in range ( X_MIN, X_MAX, GRID_SIZE ):
        for y in range ( Y_MIN, Y_MAX, GRID_SIZE ):
            domain.append ( coord_to_name ( x, y ) )

    return domain


def pyth(x0, y0, x1, y1):
    return math.sqrt ( ( x0 - x1 ) ** 2 + ( y0 - y1 ) ** 2 )


def empty_map():
    result = {}

    for cell in DOMAIN:
        result[cell] = 0

    return result


def add_maps(a, b):
    result = {}

    for cell in DOMAIN:
        result[cell] = a[cell] + b[cell]

    return result


def scale_map(density, factor):
    result = {}

    for cell in DOMAIN:
        result[cell] = density[cell] * factor

    return result


def find_max(density):
    max_density = -1
    max_cell = None

    for cell in DOMAIN:
        if density[cell] > max_density:
            max_density = density[cell]
            max_cell = cell

    if density[max_cell] == 0:
        print ( 'Warning: find_max() called on empty map' )

    coord = name_to_coord ( max_cell )
    result = { 'x': coord[0] + GRID_SIZE / 2, 'y': coord[1] + GRID_SIZE / 2 }

    return result


def calc_all_probabilities(x_mu, y_mu, sigma):
    result = {}

    for cell in DOMAIN:
        result[cell] = calc_one_probability(cell, x_mu, y_mu, sigma)

    return result


def calc_one_probability(cell, x_mu, y_mu, sigma):
    ( x_min, y_min ) = name_to_coord ( cell )
    x_max = x_min + GRID_SIZE
    y_max = y_min + GRID_SIZE

    denom = math.sqrt ( 2 ) * sigma
    x0 = ( x_min - x_mu ) / denom
    x1 = ( x_max - x_mu ) / denom
    y0 = ( y_min - y_mu ) / denom
    y1 = ( y_max - y_mu ) / denom

    ## Reduce number of calls to math.erf() in order to reduce calculation time
    ## (this does not affect accuracy in any meaningful way).

    if x0 < -3:
        erf_x0 = -1
    elif 3 < x0:
        erf_x0 = 1
    else: # -3 <= x0 <= 3
        erf_x0 = math.erf ( x0 )
```

```
    if x1 < -3:
        erf_x1 = -1
    elif 3 < x1:
        erf_x1 = 1
    else: # -3 <= x1 <= 3
        erf_x1 = math.erf ( x1 )

    if y0 < -3:
        erf_y0 = -1
    elif 3 < y0:
        erf_y0 = 1
    else: # -3 <= y0 <= 3
        erf_y0 = math.erf ( y0 )

    if y1 < -3:
        erf_y1 = -1
    elif 3 < y1:
        erf_y1 = 1
    else: # -3 <= y1 <= 3
        erf_y1 = math.erf ( y1 )

    return 0.25 * ( erf_x1 - erf_x0 ) * ( erf_y1 - erf_y0 )


def _print_map(probabilities):
    # The reversed version of "for y in range ( Y_MIN, Y_MAX, GRID_SIZE )"
    for y in range ( Y_MAX - GRID_SIZE, Y_MIN - 1, -1 * GRID_SIZE ):
        print ( '{y:04d}'.format ( y = y ), end = '\t' )
        for x in range ( X_MIN, X_MAX, GRID_SIZE ):
            cell = coord_to_name ( x, y )
            print ( '{val:6.4f}'.format ( val = probabilities[cell] ), end = '\t' )
        print ( '', end = '\n' )

    print ( '', end = '\t' )
    for x in range ( X_MIN, X_MAX, GRID_SIZE ):
        print ( '{x:04d}'.format ( x = x ), end = '\t' )
    print ( '\n', end = '\n' )


def estimate_location(detections, previous_map):
    result = scale_map ( previous_map, PREV_WEIGHT )

    for detection in detections:
        density = calc_all_probabilities ( detection['x'], detection['y'], SIGMA )
        result = add_maps ( result, density )

    return result


def analyse(experiment):
    analyse_start = datetime.datetime.now()
    print ( '' )
    print ( 'Doing init...' )

    base_filename = DATA_PATH + 'experiment{id:02d}-'.format ( id = experiment )

    csv_gt = interpolated_csv.InterpolatedCsv ( base_filename + 'ground_truth.txt' )
    csv_wall = delayed_csv.DelayedCsv ( base_filename + 'wall.txt', WALL_OFFSET )
    csv_window = delayed_csv.DelayedCsv ( base_filename + 'window.txt', WINDOW_OFFSET )

    time_start = max ( csv_gt.time_start, csv_wall.time_start, csv_window.time_start )
    time_stop = min ( csv_gt.time_stop, csv_wall.time_stop, csv_window.time_stop )

    sl_wall_map = empty_map()
    for time in range ( math.ceil ( csv_wall.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_wall_map = estimate_location ( csv_wall.get_detections ( time / 100 ), sl_wall_map )

    sl_window_map = empty_map()
    for time in range ( math.ceil ( csv_window.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_window_map = estimate_location ( csv_window.get_detections ( time / 100 ), sl_window_map )

    sl_wall_diff = 0
    sl_window_diff = 0
    samples = 0

    print ( 'Doing loop...' )
    for time in range ( math.ceil ( time_start * 100), math.floor ( time_stop * 100 ), 25 ):
        # Ground truth
        gt = csv_gt.get ( time / 100 )

        # Location as determined by subjective logic algorithm
        wall_raw = csv_wall.get_detections ( time / 100 )
        window_raw = csv_window.get_detections ( time / 100 )

        sl_wall_map = estimate_location ( wall_raw, sl_wall_map )
        sl_window_map = estimate_location ( window_raw, sl_window_map )

        sl_wall_loc = find_max ( sl_wall_map )
        sl_window_loc = find_max ( sl_window_map )

        # Calculate error
        sl_wall_diff = sl_wall_diff + pyth ( gt['x'], gt['y'], sl_wall_loc['x'], sl_wall_loc['y'] ) ** 2
        sl_window_diff = sl_window_diff + pyth ( gt['x'], gt['y'], sl_window_loc['x'], sl_window_loc['y'] ) ** 2

        samples = samples + 1

    analyse_stop = datetime.datetime.now()
```

```
    print ( '' )
    print ( 'Results_____________RMS_error' )
    print ( '' )
    print ( 'Subjective_logic_algorithm:' )
    print ( '{name:20s}__{error:8.3f}'.format ( name = 'Wall', error = math.sqrt ( sl_wall_diff / samples ) ) )
    print ( '{name:20s}__{error:8.3f}'.format ( name = 'Window', error = math.sqrt ( sl_window_diff / samples ) ) )
    print ( '' )
    print ( '' )
    print ( 'Calculation_time:', int ( (analyse_stop - analyse_start).total_seconds() ), 'seconds' )


################################################################################

# Initialise pseudo-constant
DOMAIN = _get_domain()


if len ( sys.argv ) == 1:
    print ( '\nSpecify_experiment_number_to_analyse_as_first_(and_only)_parameter.\n' )
    sys.exit()


experiment = int ( sys.argv[1] )

for weight in range ( 1, 10, 1 ):
    PREV_WEIGHT = weight / 10
    analyse ( experiment )
```

B.3.3 Optimise sigma

```
import math
import sys
import datetime

from Tools import delayed_csv
from Tools import interpolated_csv


DATA_PATH = 'data\\location\\'

WALL_OFFSET = -0.8
WINDOW_OFFSET = -8.0

GRID_SIZE = 100      ## mm
X_MIN = 2050
X_MAX = 5550
Y_MIN =  650
Y_MAX = 3750

PREV_WEIGHT = 0.7

WALL_X = 3400
WALL_Y = 4000
WINDOW_X = 1750
WINDOW_Y = 2000


# Pseudo-constant; initialised by code
DOMAIN = None


def coord_to_name(x, y):
    return 'x{x:04d}y{y:04d}'.format ( x = x, y = y )


def name_to_coord(name):
    x = int ( name[1:5] )
    y = int ( name[6:10] )

    return ( x, y )


def _get_domain():
    domain = []

    for x in range ( X_MIN, X_MAX, GRID_SIZE ):
        for y in range ( Y_MIN, Y_MAX, GRID_SIZE ):
            domain.append ( coord_to_name ( x, y ) )

    return domain


def pyth(x0, y0, x1, y1):
    return math.sqrt ( ( x0 - x1 ) ** 2 + ( y0 - y1 ) ** 2 )


def empty_map():
    result = {}

    for cell in DOMAIN:
        result[cell] = 0

    return result
```

```
def add_maps(a, b):
    result = {}

    for cell in DOMAIN:
        result[cell] = a[cell] + b[cell]

    return result


def scale_map(density, factor):
    result = {}

    for cell in DOMAIN:
        result[cell] = density[cell] * factor

    return result


def find_max(density):
    max_density = -1
    max_cell = None

    for cell in DOMAIN:
        if density[cell] > max_density:
            max_density = density[cell]
            max_cell = cell

    if density[max_cell] == 0:
        print ( 'Warning: find_max() called on empty map' )

    coord = name_to_coord ( max_cell )
    result = { 'x': coord[0] + GRID_SIZE / 2, 'y': coord[1] + GRID_SIZE / 2 }

    return result


def calc_all_probabilities(x_mu, y_mu, sigma):
    result = {}

    for cell in DOMAIN:
        result[cell] = calc_one_probability(cell, x_mu, y_mu, sigma)

    return result


def calc_one_probability(cell, x_mu, y_mu, sigma):
    ( x_min, y_min ) = name_to_coord ( cell )
    x_max = x_min + GRID_SIZE
    y_max = y_min + GRID_SIZE

    denom = math.sqrt ( 2 ) * sigma
    x0 = ( x_min - x_mu ) / denom
    x1 = ( x_max - x_mu ) / denom
    y0 = ( y_min - y_mu ) / denom
    y1 = ( y_max - y_mu ) / denom

    ## Reduce number of calls to math.erf() in order to reduce calculation time
    ## (this does not affect accuracy in any meaningful way).

    if x0 < -3:
        erf_x0 = -1
    elif 3 < x0:
        erf_x0 = 1
    else: # -3 <= x0 <= 3
        erf_x0 = math.erf ( x0 )

    if x1 < -3:
        erf_x1 = -1
    elif 3 < x1:
        erf_x1 = 1
    else: # -3 <= x1 <= 3
        erf_x1 = math.erf ( x1 )

    if y0 < -3:
        erf_y0 = -1
    elif 3 < y0:
        erf_y0 = 1
    else: # -3 <= y0 <= 3
        erf_y0 = math.erf ( y0 )

    if y1 < -3:
        erf_y1 = -1
    elif 3 < y1:
        erf_y1 = 1
    else: # -3 <= y1 <= 3
        erf_y1 = math.erf ( y1 )

    return 0.25 * ( erf_x1 - erf_x0 ) * ( erf_y1 - erf_y0 )


def _print_map(probabilities):
    # The reversed version of "for y in range ( Y_MIN, Y_MAX, GRID_SIZE )"
    for y in range ( Y_MAX - GRID_SIZE, Y_MIN - 1, -1 * GRID_SIZE ):
        print ( '{y:04d}'.format ( y = y ), end = '\t' )
        for x in range ( X_MIN, X_MAX, GRID_SIZE ):
            cell = coord_to_name ( x, y )
            print ( '{val:6.4f}'.format ( val = probabilities[cell] ), end = '\t' )
```

```
        print ( '', end = '\n' )

    print ( '', end = '\t' )
    for x in range ( X_MIN, X_MAX, GRID_SIZE ):
        print ( '{x:04d}'.format ( x = x ), end = '\t' )
    print ( '\n', end = '\n' )


def estimate_location(detections, previous_map):
    result = scale_map ( previous_map, PREV_WEIGHT )

    for detection in detections:
        density = calc_all_probabilities ( detection['x'], detection['y'], SIGMA )
        result = add_maps ( result, density )

    return result


def analyse(experiment):
    analyse_start = datetime.datetime.now()
    print ( '' )
    print ( 'Doing init...' )

    base_filename = DATA_PATH + 'experiment{id:02d}_'.format ( id = experiment )

    csv_gt = interpolated_csv.InterpolatedCsv ( base_filename + 'ground_truth.txt' )
    csv_wall = delayed_csv.DelayedCsv ( base_filename + 'wall.txt', WALL_OFFSET )
    csv_window = delayed_csv.DelayedCsv ( base_filename + 'window.txt', WINDOW_OFFSET )

    time_start = max ( csv_gt.time_start, csv_wall.time_start, csv_window.time_start )
    time_stop = min ( csv_gt.time_stop, csv_wall.time_stop, csv_window.time_stop )

    sl_wall_map = empty_map()
    for time in range ( math.ceil ( csv_wall.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_wall_map = estimate_location ( csv_wall.get_detections ( time / 100 ), sl_wall_map )

    sl_window_map = empty_map()
    for time in range ( math.ceil ( csv_window.time_start * 100), math.ceil ( time_start * 100), 25 ):
        sl_window_map = estimate_location ( csv_window.get_detections ( time / 100 ), sl_window_map )

    sl_wall_diff = 0
    sl_window_diff = 0
    samples = 0

    print ( 'Doing loop...' )
    for time in range ( math.ceil ( time_start * 100), math.floor ( time_stop * 100 ), 25 ):
        # Ground truth
        gt = csv_gt.get ( time / 100 )

        # Location as determined by subjective logic algorithm
        wall_raw = csv_wall.get_detections ( time / 100 )
        window_raw = csv_window.get_detections ( time / 100 )

        sl_wall_map = estimate_location ( wall_raw, sl_wall_map )
        sl_window_map = estimate_location ( window_raw, sl_window_map )

        sl_wall_loc = find_max ( sl_wall_map )
        sl_window_loc = find_max ( sl_window_map )

        # Calculate error
        sl_wall_diff = sl_wall_diff + pyth ( gt['x'], gt['y'], sl_wall_loc['x'], sl_wall_loc['y'] ) ** 2
        sl_window_diff = sl_window_diff + pyth ( gt['x'], gt['y'], sl_window_loc['x'], sl_window_loc['y'] ) ** 2

        samples = samples + 1

    analyse_stop = datetime.datetime.now()

    print ( '' )
    print ( 'Results_____________RMS_error' )
    print ( '' )
    print ( 'Subjective_logic_algorithm:' )
    print ( '{name:20s}__{error:8.3f}'.format ( name = 'Wall', error = math.sqrt ( sl_wall_diff / samples ) ) )
    print ( '{name:20s}__{error:8.3f}'.format ( name = 'Window', error = math.sqrt ( sl_window_diff / samples ) ) )
    print ( '' )
    print ( '' )
    print ( 'Calculation_time:', int ( (analyse_stop - analyse_start).total_seconds() ), 'seconds' )


################################################################################

# Initialise pseudo-constant
DOMAIN = _get_domain()


if len ( sys.argv ) == 1:
    print ( '\nSpecify_experiment_number_to_analyse_as_first_(and_only)_parameter.\n' )
    sys.exit()


experiment = int ( sys.argv[1] )

SIGMA = 250
analyse ( experiment )

SIGMA = 500
analyse ( experiment )

SIGMA = 750
```

```
analyse ( experiment )

SIGMA = 1000
analyse ( experiment )

SIGMA = 2000
analyse ( experiment )

SIGMA = 5000
analyse ( experiment )

SIGMA = 10000
analyse ( experiment )

SIGMA = 20000
analyse ( experiment )

SIGMA = 50000
analyse ( experiment )
```

B.3.4 Synchronise clocks

```
import math
import sys

from Tools import delayed_csv
from Tools import interpolated_csv


DATA_PATH = '..\\data\\Run_2\\'


def pyth(x0, y0, x1, y1):
    return math.sqrt ( ( x0 - x1 ) ** 2 + ( y0 - y1 ) ** 2 )


def analyse(experiment, offset):
    base_filename = DATA_PATH + 'experiment{id:02d}_'.format ( id = experiment )
    csv_gt = interpolated_csv.InterpolatedCsv ( base_filename + 'ground_truth.txt' )
    csv_wall = delayed_csv.DelayedCsv ( base_filename + 'wall.txt', offset )
    csv_window = delayed_csv.DelayedCsv ( base_filename + 'window.txt', offset )

    time_start = max ( csv_gt.time_start, csv_wall.time_start, csv_window.time_start )
    time_stop = min ( csv_gt.time_stop, csv_wall.time_stop, csv_window.time_stop )

    wall_diff = 0
    window_diff = 0
    samples = 0
    for time in range ( math.ceil ( time_start * 100), math.floor ( time_stop * 100 ), 25 ):
        gt = csv_gt.get ( time / 100 )
        wall = csv_wall.get_location ( time / 100, False )
        window = csv_window.get_location ( time / 100, False )

        wall_diff = wall_diff + pyth ( gt['x'], gt['y'], wall['x'], wall['y'] ) ** 2
        window_diff = window_diff + pyth ( gt['x'], gt['y'], window['x'], window['y'] ) ** 2
        samples = samples + 1

    print ( '{offset:6.2f}________{wall:8.2f}________{window:8.2f}'.format (
        offset = offset,
        wall = math.sqrt ( wall_diff / samples ),
        window = math.sqrt ( window_diff / samples )
    ) )



if len ( sys.argv ) == 1:
    print ( '\nSpecify experiment number to analyse as first (and only) parameter.\n' )
    sys.exit()

print ( 'Offset_________Wall____________Window' )
experiment = int ( sys.argv[1] )
for offset in range ( -100, 1, 10 ):
    analyse ( experiment, offset / 10 )
```