Exercise 1.0 — Probability and Machine Learning Basics

Artificial Intelligence For Robotics

Sebastian Verling and Tonci Novkovic

Spring Semester 2017

The goal of this exercise is to strengthen your knowledge on basics of probability and machine learning. The machine learning exercise builds up on the previous exercise. It is therefore recommended that Exercise 0.2 is solved first. An incomplete framework and the required data for this exercise can be found in the lecture's git repository: `https://github.com/ethz-asl/ai_for_robotics`.

1 Random Sampling

In this exercise you will write an algorithm to sample points from a given probability density function (PDF).

The PDF is given as follows:

$$f(x) = \frac{1}{2\pi\gamma_1\left(1+\left(\frac{x-x_{0,1}}{\gamma_1}\right)^2\right)} + \frac{1}{2\pi\gamma_2\left(1+\left(\frac{x-x_{0,2}}{\gamma_2}\right)^2\right)}$$

$$\gamma_1 = 0.3 \quad x_{0,1} = 0 \quad \gamma_2 = 0.1 \quad x_{0,2} = 3$$

Figure 1 visualizes the PDF

1.1 Approach

Remember that given a uniform sample u you can get a sample x of the probability function $f(x)$ by solving the equation $u = F(x)$ for x, where $F(x)$ is the cumulative distribution function (CDF) defined as $F(x) = \int f(x)dx$.

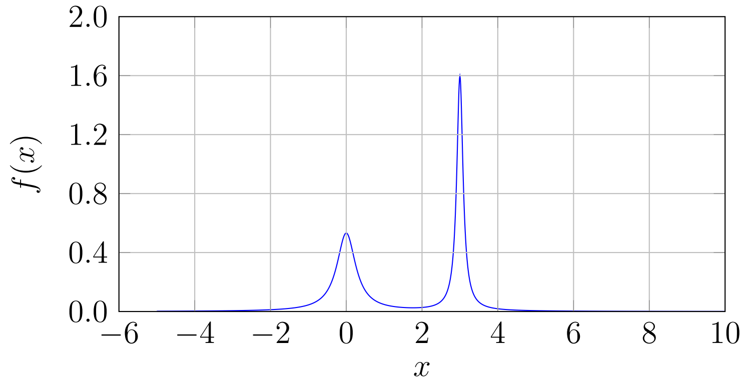

Figure 1: PDF

Hint: The equation $u = F(x)$ does not have an analytical solution for this case. In order to solve it numerically you can use the *root* function of the SciPy optimization toolbox[1].

1.2 Tasks

Fill in the sections marked with #`TODO`.

1. `main.py` fill the variable *uni_samples* with uniform samples
2. `SamplePdf.py`
   - implement the function *cdf_root*
   - implement the correct function call of the function *scipy.optimize.root*
   - calculate the pdf function values to plot the reference function

1.3 Evaluation

This task is graded as successful if the data stored in the `results.pkl` file are close to the PDF.

[1]https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.root.html

2 Bayesian Filtering

In this exercise you will apply your probability knowledge to a simple state estimation problem. Even though this exercise is showing a very simplistic case, the underlying theory is fundamental for understanding more sophisticated state estimation algorithms.

Imagine a robot moving through a corridor and collecting data about people interaction autonomously. This data is very important to the scientist using the robot, since he has this new, ground-breaking theory for modeling human behaviour which can directly be applied in economics, and if proven correct, secure his lab financing for the rest of his life. Since the scientist has more important work to do than follow the robot around the whole day, and the data is so important to him, he decided to implement a simple tracking algorithm so that he can know where the robot is at any moment, from the comfort of his office. However, since he did not take a course in state estimation, he needed to rely just on his probability experience to estimate the position of the robot.

The robot is moving in the known environment through the university corridor. The scientist walked through there millions of times before and he knew every tiny detail of it. That is why he made a map as shown in Figure 2. To simplify the problem, he discretized the corridor into 20 different positions labeled from 0 to 19 in the map, also marking positions where the doors are nearby. From his experience he knew that the doors are in fixed position in the environment and that they might be useful for tracking the robot. For this reason, he politely asked his roboticist friend from the neighbouring office to equip the robot with an RGBD camera and provide him with an algorithm to detect doors in the image. This algorithm is able to tell for each position if there is a door nearby or not. However, his friend also claimed that the algorithm is still in development and that he observed that usually one out of ten measurements is wrong. The scientist did not worry too much about this since he was confident that his filtering technique can overcome occasional wrong measurement.

The robot he deployed is always moving either 1, 2 or 3 positions in any direction (forward or backward) in one time step. However, due to a very old hardware, it can happen sometimes that robot overshoots for one position (with probability of 5%) or, more likely, undershoots for one position

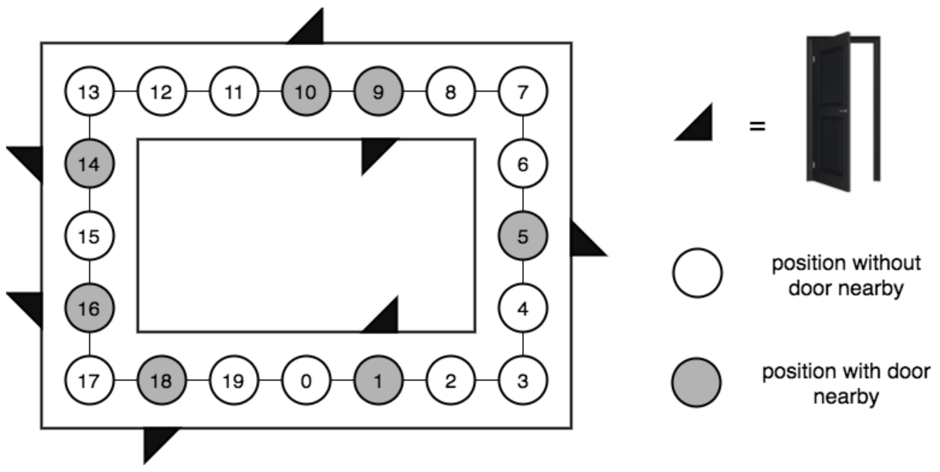

Figure 2: A map representing the simple environment robot is operating in. The positions in the map are labeled with numbers and connections between them are shown by lines. Shaded positions indicate that door is nearby.

(with probability 15%). Motion commands are computed from another very sophisticated control algorithm which he cannot change or influence in any way. Since the scientist is working remotely, he only receives the current motion command and the current measurement at each iteration. With this information he claims he can deduce the position of the robot at any given moment.

Unfortunately, since the deadline for the conference was approaching, the scientist was too busy writing the papers and he did not have enough time to implement the algorithm himself. Since his roboticist friend was away at a robotics competition in Abu Dhabi, he asked for your help. He left a set of notes to guide you through it:

- Define all the known variables
- Define the prior belief of the robot's current state
- Implement likelihood (measurement model) function:

  $$P(Z|x_i)$$

  where Z is measurement and x_i is state/position. Likelihood states how likely each position in the map is, given the current measurement and sensor noise, e.g.

```
map = [1, 0, 0]
probability_of_correct_measurement = 0.8
measurement = 1
likelihood = [0.8, 0.2, 0.2]
```

- Implement measurement update function which computes posterior probability distribution representing the probability distribution after taking the measurement into account:

  $$P(x_i|Z) = \frac{P(Z|x_i) * P(x_i)}{P(Z)}$$

  where $P(Z)$ is just a normalization term, e.g.

  ```
  prior_belief = [0.2, 0.2, 0.2]
  likelihood = [0.8, 0.2, 0.2]
  posterior_belief = [0.66, 0.17 0.17]
  ```

  Measurement update can alternatively be seen as:

  $$posterior = \frac{likelihood * prior}{normalization}$$

- Implement the prior update/prediction step which predicts the next position of the robot based on the movement of the robot and uncertainty in this measurement:

  $$P(x_i^t) = \sum_j P(x_j^{t-1}) P(x_i|x_j)$$

  where $P(x_i|x_j)$ is the probability of moving from x_j to x_i (process model) and $P(x_j^{t-1})$ is belief from the previous iteration e.g.

  ```
  movement = 1
  movement_noise_kernel = [0.1 0.8 0.1]
  posterior_belief = [0.66, 0.17 0.17]
  prior = [0.219 0.562 0.219]
  ```

- Implement the main function that performs the filtering by calling all the functions previously defined, start with the measurement update step.

Your task is to follow the scientist's notes and implement his algorithm. The robot has already been running in the corridor for some time. Your should be able to estimate the robot position based on the measurements your receive, without the knowledge of the starting position. The scientist had a little time to help you out so he left some fragments of the code which you can find on lecture's git repository: `https://github.com/ethz-asl/ai_for_robotics`. Please make sure that you use the same interface as the scientist otherwise you will not be able to estimate the robot's position. All the functions you need to implement are in *BayesianFiltering.py* file and lines you need to change are marked with #`TODO`. Please report your final state estimate (as a probability density function) of the robot.

You may have noticed that equations in the scientist's notes are nothing more than Bayes theorem for measurement update and Total probability theorem for prior update. Bayesian filtering is one of the simplest estimation techniques and there are certainly drawbacks that one needs to consider once implementing such algorithm. For example, we had a very simple 1D example here, however, with higher number of dimensions the algorithm requires significantly more computation power, often making the problem intractable. Furthermore, this type of filter is discrete. For some problems this is sufficient, but in robotics we are often dealing with continuous variables. Finally, you might have noticed that we, in addition to door measurement, required a motion measurement at each iteration. There are better methods that could be used for estimating the motion from the same sensory input (in this case RGBD camera), that are often used in robotics.

2.1 Evaluation

This task is graded as successful if the result variables stored in the `result_bayes_*.pkl` files are same as the expected PDFs.

3 Linear Regression - again

This exercise builds up on the Framework that has been worked with in the last exercise. The goal is again to identify the model of the given data set and therefore select the correct features by using crossvalidation. The main difficulty is the number of input dimensions, which makes feature selection by visual inspection significantly more difficult. Note that there are two pkl files in the data folder. The `data_samples.pkl` contains the training data you can use. The `submission_data.pkl` contains the data that is used to generate your submission. It only contains the 4 input dimensions. The file containing the results is automatically generated by the last few lines in `linear_regression.py`

Hint: The used features are combinations of the input dimensions up to second order. Additionally trigonometric functions might have been used.

3.1 The Data Points

Imagine that the given 400 Data Points are gathered from 4 different experiment sessions where each time 100 Data Points were measured. As you heard in the lecture this can mean that the Points may have different bias and noise values. Think about a solution to overcome this problem. You can assume that the first 100 data points come from the first experiment and the next 100 from the second experiment and so on.

3.2 Tasks

In general, fill in the sections marked with #`TODO` in order to get the best possible linear regression.

The main tasks are

1. `DataSaver.py`: adapt data loader to have it working with 4 input dimensions
2. `Features.py`: extend the list of features with the ones that you consider to be useful when looking at the data

3. `LinearRegressionModel.py`: adapt it such that it works with 4 input dimensions
4. `LinearRegressionModel.py`: think about a solution to the problem described in Subsection 3.1 (differently biased Data)

3.3 Evaluation

The grade of this task will be based on the mean squared error (MSE) of your predicted data in the `results.pkl` file, which is automatically generated

4 Submission

To hand in the exercise you have to zip the whole folder (`1_0_probability_ml_basics`) with your code and results files and upload it on moodle. The zip file you upload should have the name *aifr_familyname_ex1* (replace familyname with your family name). The python scripts are written in a way, such that it always creates a `results.pkl` file with the data that is used for evaluation.