Demo

August 7, 2020

1 BasicBayes Tutorial

1.1 Intro

The module has 3 parts:

1. **Parameters**: Parameters are objects which are used to specify probability distributions. Examples include the mean and standard deviation parameters of gaussian distributions.
2. **Estimators**: Estimators are objects which take parameters, data, and a likelihood function, and find the values of the parameters that most probably explain the data under a given likelihood function.
3. **Likelihood Functions**: In general, BasicBayes requires you to write your own likelihood function when using an Estimator. After all, there are way too many possibilities for how you might want to calculate your likelihoods, for this module to include them all. However, BasicBayes does include a library of commonly-used likelihood functions.

The general process for conducting Bayesian Analysis within Basic Bayes involves creating your parameters, specifying their relevant hypotheses and priors, creating/choosing a likelihood function, and finally fitting an estimator to your data.

1.2 Parameters

The parameter class has the following API:

1.2.1 Methods

- `Parameter(name, description)`: Creates and outputs a parameter object, with the given name and description.
  - *`name`* (Optional): The name for the given parameter. It will be used to title the plot of this parameter's PDF. If not given, it defaults to "Parameter".
  - *`description`* (Optional): The description for the given parameter. It will be used to caption the plot of this parameter's PDF. If not given, it defaults to "A Parameter".
- `bin(start, stop, num_steps, step, skip_first)`: Creates the hypotheses for the parameter. Use this to assign hypotheses for a numeric parameter. This also assigns a flat prior.
  - *`start`*: The lower bound on the parameter's value.
  - *`stop`*: The upper bound on the parameter's value.

  - *num_steps* (Optional if `step` provided): The number of samples taken as hypotheses in the range [start, stop]
  - *step* (Optional if `num_steps` provided): The step between samples taken as hypotheses in the range [start, stop]
  - *skip_first* (Optional): Whether or not to skip the first hypothesis, useful if you want a range for the parameter hypotheses that is open at the lower bound (i.e. (start, stop]). Defaults to `False`.

- `categorical_bin(categories)`: Assigns the hypotheses for parameters that are categorical, rather than numerical.
  - *categories*: The categorical values of the parameter, as a list. For example, this could be `["Heads", "Tails"]` if the parameter is describing the outcome of a coin flip.
- `plot_pdf()`: Plots the PDF for the parameter in a new window.
- `mode()`: Returns the mode of the PDF
- `CI(probability)`: Returns the confidence interval for the parameter's value as a tuple.
  - *probability* (Optional): The level of confidence for which the confidence interval is returned. Defaults to 0.95.
- `set_probabilities(function)`: Sets the pobabilities for each hypothesis. This can be used to set up priors.
  - *function*: The function to be used to assign a probability to each hypothesis.
- `Comparison Operators <, >, <=, >=, ==`: The comparison operators are overridden so that applying them to any 2 parameters returns a probability, rather than a boolean. For example, `1 < 2` returns `False`, but `parameter_1 < parameter_2` returns the probability that parameter_1 is less than parameter_2

1.2.2 Attributes

- `bins`: An array that contains the hypothesis values.
- `probabilities`: The probability for the respective hypothesis in `bins`.
- `type`: A string describing whether the parameter is "Numeric" or "Categorical"
- `name`: The name of the parameter.
- `description`: The description of the parameter.

1.3 Estimators

Estimator objects have the following API:

1.3.1 Methods

- `Estimator(self, function, parameters, name, description)`: Creates and returns an estimator object.
  - `function`: The likelihood function used to fit the data.

  - `parameters`: The list of parameters whose values are used in the likelihood function, in the order that they need to be provided to the likelihood function. For example, if the likelihood function is `gaussian(x, mu, sigma)`, `parameters` should be the list `[mu_parameter, sigma_parameter]`.
  - *`name`* (Optional): The name for the given parameter. It will be used to title the plot of this parameter's PDF. If not given, it defaults to "Parameter".
  - *`description`* (Optional): The description for the given parameter. It will be used to caption the plot of this parameter's PDF. If not given, it defaults to "A Parameter".
- `fit(data, mcmc, return_mode)`: Uses Bayesian Inference to find the parameter values that best explain `data`. After calling this function, the parameters in the estimator will have their probability distributions modified based on data, and the mode of the joint posterior PDF is returned.
  - *`data`*: An iterable collection (ie. list, tuple, or array) of data observations.
  - *`mcmc`* (Optional): Whether or not to use Markov-Chain Monte-Carlo to approximate the posterior PDF. Defaults to `False`.
- `set_probabilities(function)`: Sets the probabilities of the estimator's internal joint PDF. Can be used to establish a prior.
  - *`function`*: A function of the different parameters that make up the estimator, giving the probability for each combination of parameters.
- `Calling the Estimator object like a function`: The estimator object itself can be called on a single datapoint, an iterable collection of data points, or a parameter to determine the probability of observing that data point, that series of data points, or any of the values in the parameter's hypothesis range, respectively.

1.3.2 Attributes

- `function`: The likelihood function used by this estimator.
- `parameters`: The list of parameter objects that are used by the likelihood function.
- `name`: The name of the estimator.
- `description`: The description of the estimator.
- `probabilities`: The joint PDF used by the estimator

1.4 Likelihood Function Library

1.5 Examples

1.5.1 Fitting a Gaussian Distribution: Numeric Data, Numeric Parameters

Let's say we have a series of data points, and we want to find the gaussian distribution that is the most likely to get us these observations, if we drew randomly from it. Suppose our data is the following list of observations:

[8]:
```
X = [-1, -2, -1, -3, 2, 1, 0, 1]
```

To find the best fit, wwe can use the following workflow:

1. Create the mean and standard deviation parameters.
2. Create a likelihood function, which in this case is just the gaussian distribution and comes pre-loaded as part of BasicBayes.
3. Create the estimator, and fit it to the data.
4. Find the modes of the marginalized PDFs for the mean and stdev.

[9]:
```
from BasicBayes import Parameter, Estimator

# Create the parameters and "bin" them to make the hypotheses
mu = Parameter(name="Mu", description="Gaussian mean")
mu.bin(min(X), max(X), 100)
sigma = Parameter(name="Sigma", description="Gaussian standard deviation")
sigma.bin(0, max(X) - min(X), 100, skip_first=True)

# Get the likelihood function
from BasicBayes.LikelihoodFunctions import gaussian

# Create the estimator, and fit the data
gaussian_estimator = Estimator(function=gaussian, parameters=[mu, sigma])
gaussian_estimator.fit(X)

# Plot the PDFs, and get the mode
mode = gaussian_estimator.mode()
print(f"The best fit gaussian has a mean of {mode[0]} and a standard deviation␣
 ↪of {mode[1]}")
```

```
The best fit gaussian has a mean of -0.3737373737373737 and a standard deviation
of 1.6
```

In case we only cared about the mean of the optimal gaussian distribution, we might want to marginalize across the standard deviation. To get the marhinalized pdf, it's really simple. All we have to do is ask the mu parameter for its mode and 95% CI, rather than the estimator. This works because the `Estimator.fit()` function takes care of marginalization for us, by changing the PDFs of the parameters in the Estimator to their marginalized PDFs.

[10]:
```
print(f"The mean of the population from which the data was drawn has mode {mu.
 ↪mode()} with a 95% CI of {mu.CI(0.95)}.")
```

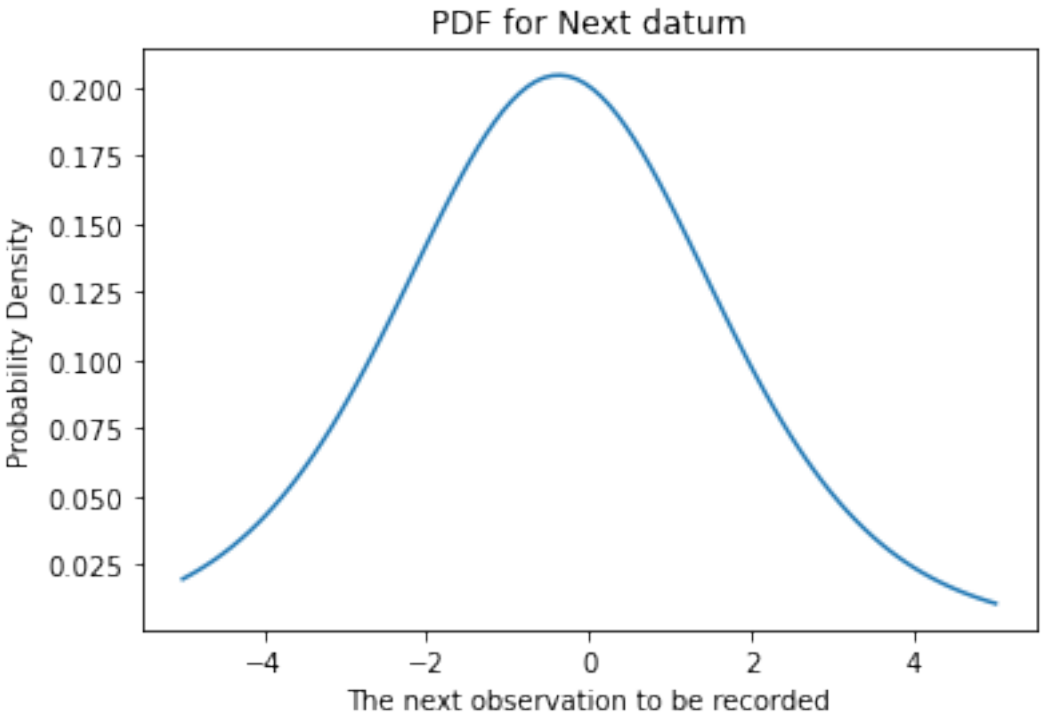
```
The mean of the population from which the data was drawn has mode
-0.3737373737373737 with a 95% CI of (-1.8888888888888888, 1.0909090909090908).
```

Now that we have the fitted estimator, we can even get the probability distribution for the next observation. To do this, we can set up a parameter to represent this next observation, and call the estimator on it. In this case, you can think of the estimator itself as a function which outputs the probabilities for any value

[11]:
```
# Create the parameter for the next observation (don't forget to bin it)
next_datum = Parameter(name="Next datum", description="The next observation to␣
 ↪be recorded").bin(-5, 5, 100)

# Call the estimator on it to transform it
next_datum = gaussian_estimator(next_datum)

# view the PDF
next_datum.plot_pdf()
print(f"The most likely next observation is {next_datum.mode()}, with 95%␣
 ↪confidence interval {next_datum.CI(0.95)}.")
```




```
The most likely next observation is -0.3535353535353538, with 95% confidence
interval (-4.292929292929293, 3.4848484848484844).
```

1.5.2 Bias of a Coin: Categorical Data, Numeric Parameters

Suppose we have a coin, which we have reason to believe is rigged to land heads with a probability other than 50%. We observe the following sequence of flips:

[12]:
```
X = ["H", "H", "H", "H", "T", "T", "H", "T", "H"]
```

To estimate the bias on the coin, as well as our uncertainty about it, we will follow the following workflow:

1. Set up a parameter representing the coin's bias
2. Create the likelihood function
3. Create the estimator, and fit the data
4. Check the PDF for the bias parameter

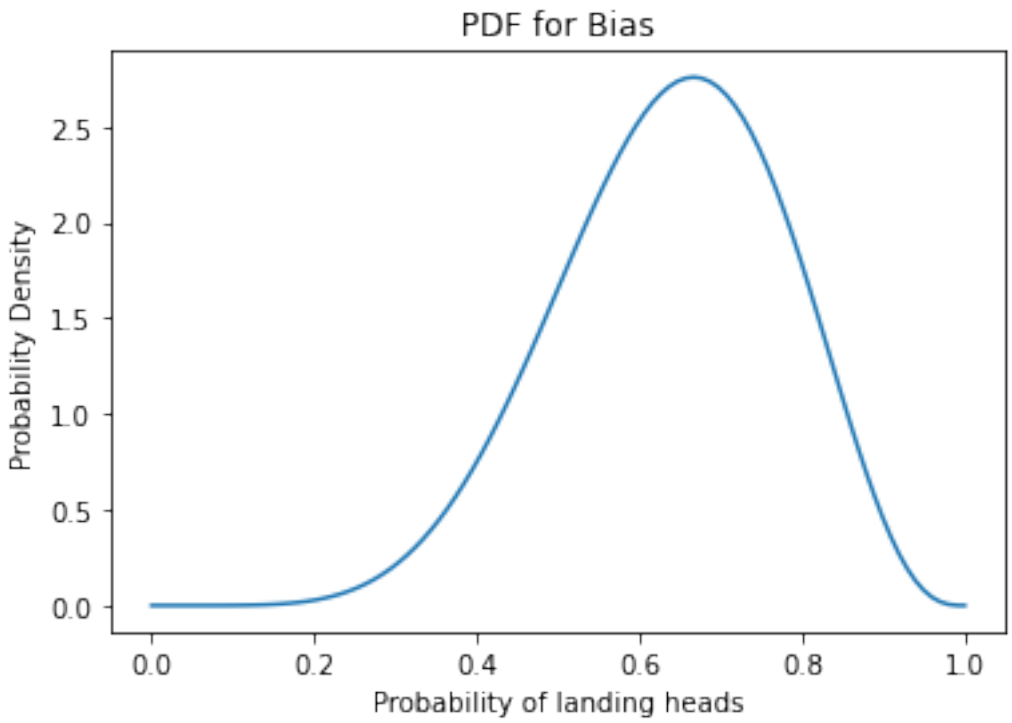
[14]:
```
from BasicBayes import Parameter, Estimator

# Set up a parameter representing the coin's bias
bias = Parameter("Bias", "Probability of landing heads").bin(0, 1, 100)

# Set up the likelihood function
def likelihood_of_flip(flip_outcome, bias):
    if flip_outcome == "H":
        return bias
    elif flip_outcome == "T":
        return 1 - bias

# Create the estimator, fit the data
bias_estimator = Estimator(function=likelihood_of_flip, parameters=[bias],␣
↪name="Bias Estimator", description="Estimation for the bias of the coin")
bias_estimator.fit(X)

# Check the PDF for the bias parameter
bias.plot_pdf()
print(f"The bias of the coin is {bias.mode()} with a 95% CI of {bias.CI(0.95)}.
↪")
```




```
The bias of the coin is 0.6666666666666667 with a 95% CI of
(0.36363636363636365, 0.888888888888889).
```

NOTE: In this case, since the estimator is built using only 1 parameter, the "joint PDF" it stores internally is the exact same as the "marginalized PDF" stored in the bias parameter, and we can check either one to get the info we're after.

1.5.3 Estimating Specificity and Sensitivity of a Test

Suppose we have 100 patients, and we're screening them for Senioritis using both a new test, and the gold standard test which is independent in terms of the physiological traits it screens. After screening them, we get the following confusion matrix:

	Positive Test	Negative Test
Actually Positive	41	9
Actually Negative	20	30

Although we can tell just from the raw data that our sensitivity = 41/(41+9) = 0.82, and our specificity = 30/(20+30) = 0.6, we may still want the posteriors to get an idea of our uncertainty. Since our study only used a sample of 100, we also want to account for the fact that the prevalence of Senioritis in our study is an estimate of the true prevalence in our population. If you didn't see right away that we needed to introduce a new parameter, you would have run into it (like I did) while writing your likelihood function. For that, we need to know P(Actually Positive) and P(Actually Negative). While we could use the proportion of positive and negative cases in our

study as these values, a more accurate way would be to treat the prevalence as another unknown parameter, with its own PDF based on our data.

[2]:
```
from BasicBayes import Parameter, Estimator

# In this case, each data
# point is a confusion matrix, and we just have one.
# Represent the confusion matrix as a python array
import numpy as np
X = np.array([[41, 9],[20, 30]])

# Create the parameters
specificity = Parameter("Specificity", "Specificity for Senioritis Test").
 ↪bin(0, 1, 100)
sensitivity = Parameter("Sensitivity", "Sensitivity for Senioritis Test").
 ↪bin(0, 1, 100)
prevalence = Parameter("Prevalence", "Prevalence of Senioritis").bin(0, 1, 100)

# Create the likelihood function for the data, given a value for sensitivity␣
 ↪and specificity
def likelihood(confusion_matrix, specificity, sensitivity, prevalence):
    true_positives_probability = prevalence*sensitivity
    false_positives_probability = prevalence*(1-sensitivity)
    true_negatives_probability = (1-prevalence)*specificity
    false_negatives_probability = (1-prevalence)*(1-specificity)
    n_TP = confusion_matrix[0, 0]
    n_FP = confusion_matrix[0, 1]
    n_TN = confusion_matrix[1, 1]
    n_FN = confusion_matrix[1, 0]

    true_positives_likelihood = true_positives_probability**n_TP
    false_positives_likelihood = false_positives_probability**n_FP
    true_negatives_likelihood = true_negatives_probability**n_TN
    false_negatives_likelihood = false_negatives_probability**n_FN

    return␣
 ↪true_positives_likelihood*false_positives_likelihood*true_negatives_likelihood*false_negati

# Create the estimator
test_estimator = Estimator(likelihood, parameters=[specificity, sensitivity,␣
 ↪prevalence])
test_estimator.fit(X)

# See the PDFs
mode = test_estimator.mode()
print(f"The prevalence of Senioritis is {prevalence.mode()} with a 95% CI of␣
 ↪{prevalence.CI(0.95)}.")
```

```
print(f"The specificity of the test is {specificity.mode()} with a 95% CI of␣
 ↪{specificity.CI(0.95)}.")
print(f"The sensitivity of the test is {sensitivity.mode()} with a 95% CI of␣
 ↪{sensitivity.CI(0.95)}.")
```

```
The prevalence of Senioritis is 0.51 with a 95% CI of (0.4, 0.6).
The specificity of the test is 0.6 with a 95% CI of (0.46, 0.73).
The sensitivity of the test is 0.82 with a 95% CI of (0.71, 0.91).
```

[ ]: