Ch6_Priors_PyMC2

January 4, 2018

1 Chapter 6

This chapter of Bayesian Methods for Hackers focuses on the most debated and discussed part of Bayesian methodologies: how to choose an appropriate prior distribution. We also present how the prior's influence changes as our dataset increases, and an interesting relationship between priors and penalties on linear regression.

1.1 Getting our priorities straight

Up until now, we have mostly ignored our choice of priors. This is unfortunate as we can be very expressive with our priors, but we also must be careful about choosing them. This is especially true if we want to be objective, that is, not to express any personal beliefs in the priors.

1.1.1 Subjective vs Objective priors

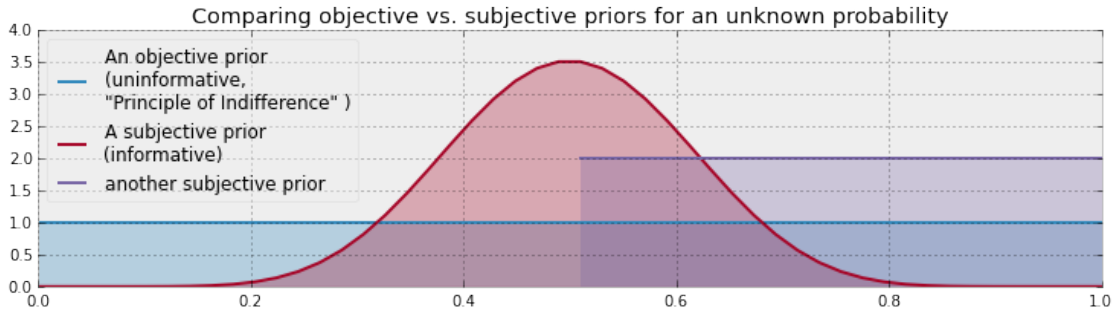
Bayesian priors can be classified into two classes: *objective* priors, which aim to allow the data to influence the posterior the most, and *subjective* priors, which allow the practitioner to express his or her views into the prior.

What is an example of an objective prior? We have seen some already, including the *flat* prior, which is a uniform distribution over the entire possible range of the unknown. Using a flat prior implies that we give each possible value an equal weighting. Choosing this type of prior is invoking what is called "The Principle of Indifference", literally we have no prior reason to favor one value over another. Calling a flat prior over a restricted space an objective prior is not correct, though it seems similar. If we know p in a Binomial model is greater than 0.5, then $\text{Uniform}(0.5, 1)$ is not an objective prior (since we have used prior knowledge) even though it is "flat" over [0.5, 1]. The flat prior must be flat along the *entire* range of possibilities.

Aside from the flat prior, other examples of objective priors are less obvious, but they contain important characteristics that reflect objectivity. For now, it should be said that *rarely* is a objective prior *truly* objective. We will see this later.

Subjective Priors On the other hand, if we added more probability mass to certain areas of the prior, and less elsewhere, we are biasing our inference towards the unknowns existing in the former area. This is known as a subjective, or *informative* prior. In the figure below, the subjective prior reflects a belief that the unknown likely lives around 0.5, and not around the extremes. The objective prior is insensitive to this.

```
In [1]: %matplotlib inline
        import numpy as np
        from IPython.core.pylabtools import figsize
        import matplotlib.pyplot as plt
        import scipy.stats as stats

        figsize(12.5, 3)
        colors = ["#348ABD", "#A60628", "#7A68A6", "#467821"]

        x = np.linspace(0, 1)
        y1, y2 = stats.beta.pdf(x, 1, 1), stats.beta.pdf(x, 10, 10)

        p = plt.plot(x, y1,
                     label='An objective prior \n(uninformative, \n"Principle of Indifference"
        plt.fill_between(x, 0, y1, color=p[0].get_color(), alpha=0.3)

        p = plt.plot(x, y2,
                     label="A subjective prior \n(informative)")
        plt.fill_between(x, 0, y2, color=p[0].get_color(), alpha=0.3)

        p = plt.plot(x[25:], 2 * np.ones(25), label="another subjective prior")
        plt.fill_between(x[25:], 0, 2, color=p[0].get_color(), alpha=0.3)

        plt.ylim(0, 4)

        plt.ylim(0, 4)
        leg = plt.legend(loc="upper left")
        leg.get_frame().set_alpha(0.4)
        plt.title("Comparing objective vs. subjective priors for an unknown probability");
```



The choice of a subjective prior does not always imply that we are using the practitioner's subjective opinion: more often the subjective prior was once a posterior to a previous problem, and now the practitioner is updating this posterior with new data. A subjective prior can also be used to inject *domain knowledge* of the problem into the model. We will see examples of these two situations later.

1.1.2 Decision, decisions...

The choice, either *objective* or *subjective* mostly depends on the problem being solved, but there are a few cases where one is preferred over the other. In instances of scientific research, the choice of an objective prior is obvious. This eliminates any biases in the results, and two researchers who might have differing prior opinions would feel an objective prior is fair. Consider a more extreme situation:

> A tobacco company publishes a report with a Bayesian methodology that retreated 60 years of medical research on tobacco use. Would you believe the results? Unlikely. The researchers probably chose a subjective prior that too strongly biased results in their favor.

Unfortunately, choosing an objective prior is not as simple as selecting a flat prior, and even today the problem is still not completely solved. The problem with naively choosing the uniform prior is that pathological issues can arise. Some of these issues are pedantic, but we delay more serious issues to the Appendix of this Chapter (TODO).

We must remember that choosing a prior, whether subjective or objective, is still part of the modeling process. To quote Gelman [5]:

> ...after the model has been fit, one should look at the posterior distribution and see if it makes sense. If the posterior distribution does not make sense, this implies that additional prior knowledge is available that has not been included in the model, and that contradicts the assumptions of the prior distribution that has been used. It is then appropriate to go back and alter the prior distribution to be more consistent with this external knowledge.

If the posterior does not make sense, then clearly one had an idea what the posterior *should* look like (not what one *hopes* it looks like), implying that the current prior does not contain all the prior information and should be updated. At this point, we can discard the current prior and choose a more reflective one.

Gelman [4] suggests that using a uniform distribution with large bounds is often a good choice for objective priors. Although, one should be wary about using Uniform objective priors with large bounds, as they can assign too large of a prior probability to non-intuitive points. Ask yourself: do you really think the unknown could be incredibly large? Often quantities are naturally biased towards 0. A Normal random variable with large variance (small precision) might be a better choice, or an Exponential with a fat tail in the strictly positive (or negative) case.

If using a particularly subjective prior, it is your responsibility to be able to explain the choice of that prior, else you are no better than the tobacco company's guilty parties.

1.1.3 Empirical Bayes

While not a true Bayesian method, *empirical Bayes* is a trick that combines frequentist and Bayesian inference. As mentioned previously, for (almost) every inference problem there is a Bayesian method and a frequentist method. The significant difference between the two is that Bayesian methods have a prior distribution, with hyperparameters α, while empirical methods do not have any notion of a prior. Empirical Bayes combines the two methods by using frequentist methods to select α, and then proceeds with Bayesian methods on the original problem.

A very simple example follows: suppose we wish to estimate the parameter μ of a Normal distribution, with $\sigma = 5$. Since μ could range over the whole real line, we can use a Normal

distribution as a prior for μ. How to select the prior's hyperparameters, denoted (μ_p, σ_p^2)? The σ_p^2 parameter can be chosen to reflect the uncertainty we have. For μ_p, we have two options:

1. Empirical Bayes suggests using the empirical sample mean, which will center the prior around the observed empirical mean:

$$\mu_p = \frac{1}{N}\sum_{i=0}^{N} X_i$$

2. Traditional Bayesian inference suggests using prior knowledge, or a more objective prior (zero mean and fat standard deviation).

Empirical Bayes can be argued as being semi-objective, since while the choice of prior model is ours (hence subjective), the parameters are solely determined by the data.

Personally, I feel that Empirical Bayes is *double-counting* the data. That is, we are using the data twice: once in the prior, which will influence our results towards the observed data, and again in the inferential engine of MCMC. This double-counting will understate our true uncertainty. To minimize this double-counting, I would only suggest using Empirical Bayes when you have *lots* of observations, else the prior will have too strong of an influence. I would also recommend, if possible, to maintain high uncertainty (either by setting a large σ_p^2 or equivalent.)

Empirical Bayes also violates a theoretical axiom in Bayesian inference. The textbook Bayesian algorithm of:

$$\textit{prior} \Rightarrow \textit{observed data} \Rightarrow \textit{posterior}$$

is violated by Empirical Bayes, which instead uses

$$\textit{observed data} \Rightarrow \textit{prior} \Rightarrow \textit{observed data} \Rightarrow \textit{posterior}$$

Ideally, all priors should be specified *before* we observe the data, so that the data does not influence our prior opinions (see the volumes of research by Daniel Kahneman *et. al* about anchoring).

1.2 Useful priors to know about

1.2.1 The Gamma distribution

A Gamma random variable, denoted $X \sim \text{Gamma}(\alpha, \beta)$, is a random variable over the positive real numbers. It is in fact a generalization of the Exponential random variable, that is:

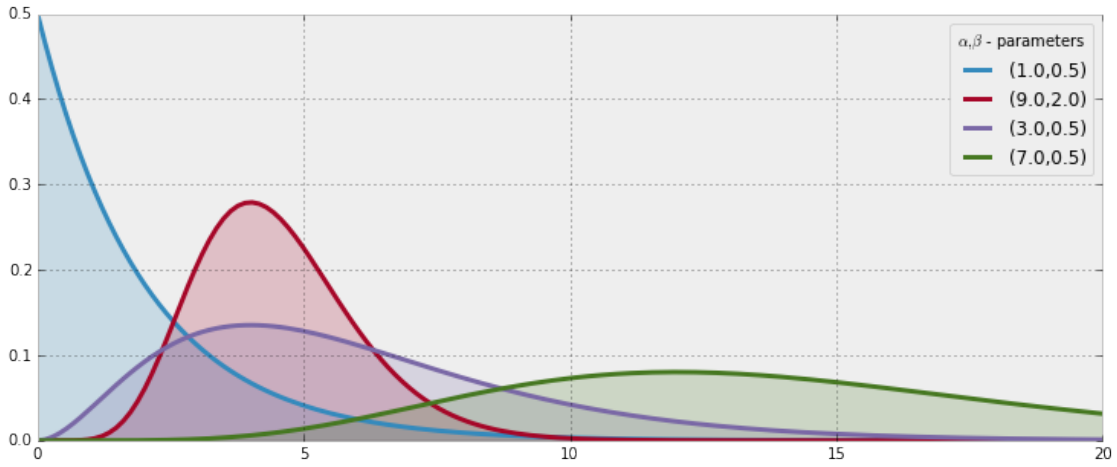
$$\text{Exp}(\beta) \sim \text{Gamma}(1, \beta)$$

This additional parameter allows the probability density function to have more flexibility, hence allowing the practitioner to express his or her subjective priors more accurately. The density function for a $\text{Gamma}(\alpha, \beta)$ random variable is:

$$f(x \mid \alpha, \beta) = \frac{\beta^{\alpha} x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

where $\Gamma(\alpha)$ is the Gamma function, and for differing values of (α, β) looks like:

```
In [2]: figsize(12.5, 5)
        gamma = stats.gamma

        parameters = [(1, 0.5), (9, 2), (3, 0.5), (7, 0.5)]
        x = np.linspace(0.001, 20, 150)
        for alpha, beta in parameters:
            y = gamma.pdf(x, alpha, scale=1. / beta)
            lines = plt.plot(x, y, label="(%.1f,%.1f)" % (alpha, beta), lw=3)
            plt.fill_between(x, 0, y, alpha=0.2, color=lines[0].get_color())
            plt.autoscale(tight=True)

        plt.legend(title=r"$\alpha, \beta$ - parameters");
```

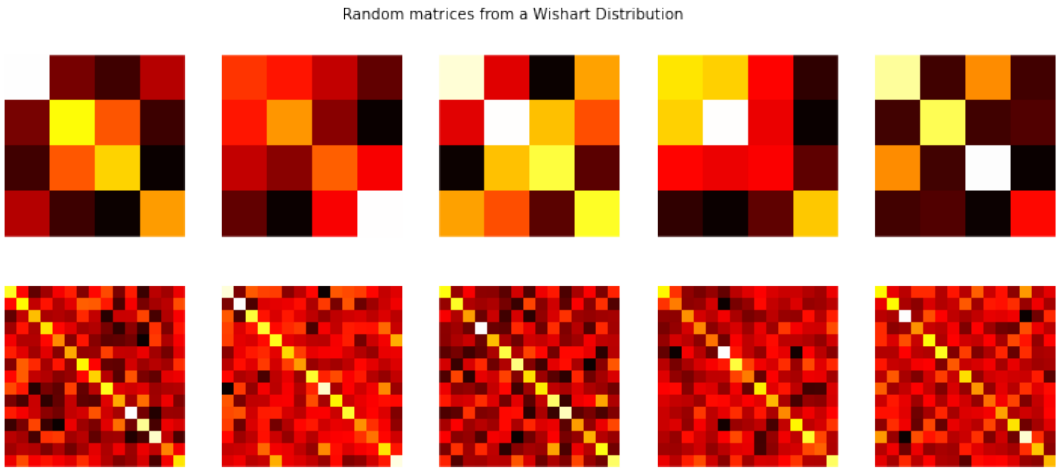


1.2.2 The Wishart distribution

Until now, we have only seen random variables that are scalars. Of course, we can also have *random matrices*! Specifically, the Wishart distribution is a distribution over all positive semi-definite matrices. Why is this useful to have in our arsenal? (Proper) covariance matrices are positive-definite, hence the Wishart is an appropriate prior for covariance matrices. We can't really visualize a distribution of matrices, so I'll plot some realizations from the 5×5 (above) and 20×20 (below) Wishart distribution:

```
In [3]: import pymc as pm

        n = 4
        for i in range(10):
            ax = plt.subplot(2, 5, i + 1)
            if i >= 5:
                n = 15
            plt.imshow(pm.rwishart(n + 1, np.eye(n)), interpolation="none",
                       cmap=plt.cm.hot)
```

```
    ax.axis("off")

plt.suptitle("Random matrices from a Wishart Distribution");
```



One thing to notice is the symmetry of these matrices. The Wishart distribution can be a little troubling to deal with, but we will use it in an example later.

1.2.3 The Beta distribution

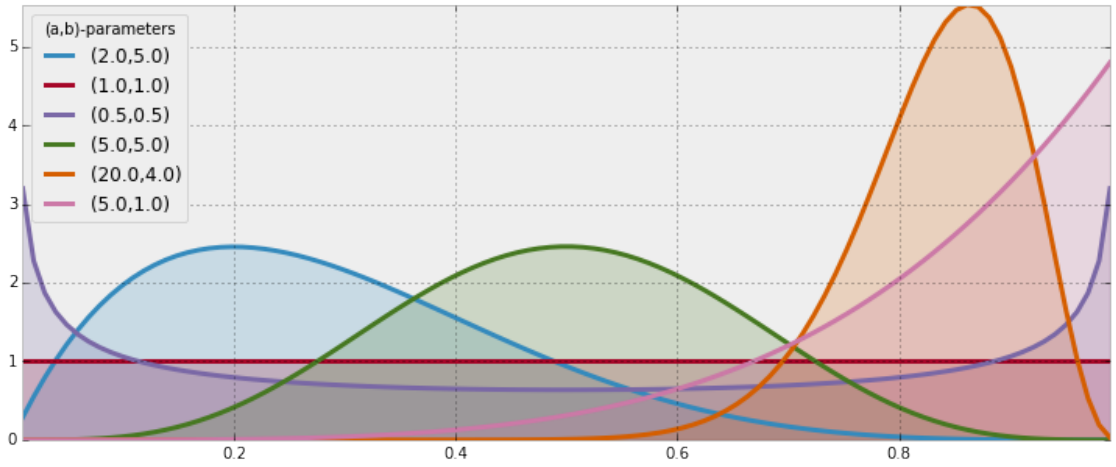
You may have seen the term `beta` in previous code in this book. Often, I was implementing a Beta distribution. The Beta distribution is very useful in Bayesian statistics. A random variable X has a Beta distribution, with parameters (α, β), if its density function is:

$$f_X(x|\ \alpha, \beta) = \frac{x^{(\alpha-1)}(1-x)^{(\beta-1)}}{B(\alpha, \beta)}$$

where B is the Beta function (hence the name). The random variable X is only allowed in [0,1], making the Beta distribution a popular distribution for decimal values, probabilities and proportions. The values of α and β, both positive values, provide great flexibility in the shape of the distribution. Below we plot some distributions:

```
In [4]: figsize(12.5, 5)

        params = [(2, 5), (1, 1), (0.5, 0.5), (5, 5), (20, 4), (5, 1)]

        x = np.linspace(0.01, .99, 100)
        beta = stats.beta
        for a, b in params:
            y = beta.pdf(x, a, b)
            lines = plt.plot(x, y, label="(%.1f,%.1f)" % (a, b), lw=3)
            plt.fill_between(x, 0, y, alpha=0.2, color=lines[0].get_color())
            plt.autoscale(tight=True)
```

```
plt.ylim(0)
plt.legend(loc='upper left', title="(a,b)-parameters");
```



One thing I'd like the reader to notice is the presence of the flat distribution above, specified by parameters $(1,1)$. This is the Uniform distribution. Hence the Beta distribution is a generalization of the Uniform distribution, something we will revisit many times.

There is an interesting connection between the Beta distribution and the Binomial distribution. Suppose we are interested in some unknown proportion or probability p. We assign a Beta(α, β) prior to p. We observe some data generated by a Binomial process, say $X \sim$ Binomial(N, p), with p still unknown. Then our posterior *is again a Beta distribution*, i.e. $p|X \sim$ Beta$(\alpha + X, \beta + N - X)$. Succinctly, one can relate the two by "a Beta prior with Binomial observations creates a Beta posterior". This is a very useful property, both computationally and heuristically.

In light of the above two paragraphs, if we start with a Beta$(1,1)$ prior on p (which is a Uniform), observe data $X \sim$ Binomial(N, p), then our posterior is Beta$(1 + X, 1 + N - X)$.

Example: Bayesian Multi-Armed Bandits *Adapted from an example by Ted Dunning of MapR Technologies*

> Suppose you are faced with N slot machines (colourfully called multi-armed bandits). Each bandit has an unknown probability of distributing a prize (assume for now the prizes are the same for each bandit, only the probabilities differ). Some bandits are very generous, others not so much. Of course, you don't know what these probabilities are. By only choosing one bandit per round, our task is devise a strategy to maximize our winnings.

Of course, if we knew the bandit with the largest probability, then always picking this bandit would yield the maximum winnings. So our task can be phrased as "Find the best bandit, and as quickly as possible".

The task is complicated by the stochastic nature of the bandits. A suboptimal bandit can return many winnings, purely by chance, which would make us believe that it is a very profitable bandit. Similarly, the best bandit can return many duds. Should we keep trying losers then, or give up?

A more troublesome problem is, if we have a found a bandit that returns *pretty good* results, do we keep drawing from it to maintain our *pretty good score*, or do we try other bandits in hopes of finding an *even-better* bandit? This is the exploration vs. exploitation dilemma.

1.2.4 Applications

The Multi-Armed Bandit problem at first seems very artificial, something only a mathematician would love, but that is only before we address some applications:

- Internet display advertising: companies have a suite of potential ads they can display to visitors, but the company is not sure which ad strategy to follow to maximize sales. This is similar to A/B testing, but has the added advantage of naturally minimizing strategies that do not work (and generalizes to A/B/C/D... strategies)
- Ecology: animals have a finite amount of energy to expend, and following certain behaviours has uncertain rewards. How does the animal maximize its fitness?
- Finance: which stock option gives the highest return, under time-varying return profiles.
- Clinical trials: a researcher would like to find the best treatment, out of many possible treatment, while minimizing losses.
- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

Many of these questions above are fundamental to the application's field.

It turns out the *optimal solution* is incredibly difficult, and it took decades for an overall solution to develop. There are also many approximately-optimal solutions which are quite good. The one I wish to discuss is one of the few solutions that can scale incredibly well. The solution is known as *Bayesian Bandits*.

1.2.5 A Proposed Solution

Any proposed strategy is called an *online algorithm* (not in the internet sense, but in the continuously-being-updated sense), and more specifically a reinforcement learning algorithm. The algorithm starts in an ignorant state, where it knows nothing, and begins to acquire data by testing the system. As it acquires data and results, it learns what the best and worst behaviours are (in this case, it learns which bandit is the best). With this in mind, perhaps we can add an additional application of the Multi-Armed Bandit problem:

- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

The Bayesian solution begins by assuming priors on the probability of winning for each bandit. In our vignette we assumed complete ignorance of these probabilities. So a very natural prior is the flat prior over 0 to 1. The algorithm proceeds as follows:

For each round:

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select $B = \text{argmax } X_b$.
3. Observe the result of pulling bandit B, and update your prior on bandit B.
4. Return to 1.

That's it. Computationally, the algorithm involves sampling from N distributions. Since the initial priors are Beta$(\alpha = 1, \beta = 1)$ (a uniform distribution), and the observed result X (a win or loss, encoded 1 and 0 respectfully) is Binomial, the posterior is a Beta$(\alpha = 1 + X, \beta = 1 + 1X)$.

To answer our question from before, this algorithm suggests that we should not discard losers, but we should pick them at a decreasing rate as we gather confidence that there exist *better* bandits. This follows because there is always a non-zero chance that a loser will achieve the status of B, but the probability of this event decreases as we play more rounds (see figure below).

Below we implement Bayesian Bandits using two classes, `Bandits` that defines the slot machines, and `BayesianStrategy` which implements the above learning strategy.

```
In [5]: from pymc import rbeta
```

```
        class Bandits(object):

            """
            This class represents N bandits machines.

            parameters:
                p_array: a (n,) Numpy array of probabilities >0, <1.

            methods:
                pull( i ): return the results, 0 or 1, of pulling
                           the ith bandit.
            """

            def __init__(self, p_array):
                self.p = p_array
                self.optimal = np.argmax(p_array)

            def pull(self, i):
                # i is which arm to pull
                return np.random.rand() < self.p[i]

            def __len__(self):
                return len(self.p)


        class BayesianStrategy(object):

            """
            Implements a online, learning strategy to solve
            the Multi-Armed Bandit problem.

            parameters:
                bandits: a Bandit class with .pull method

            methods:
```

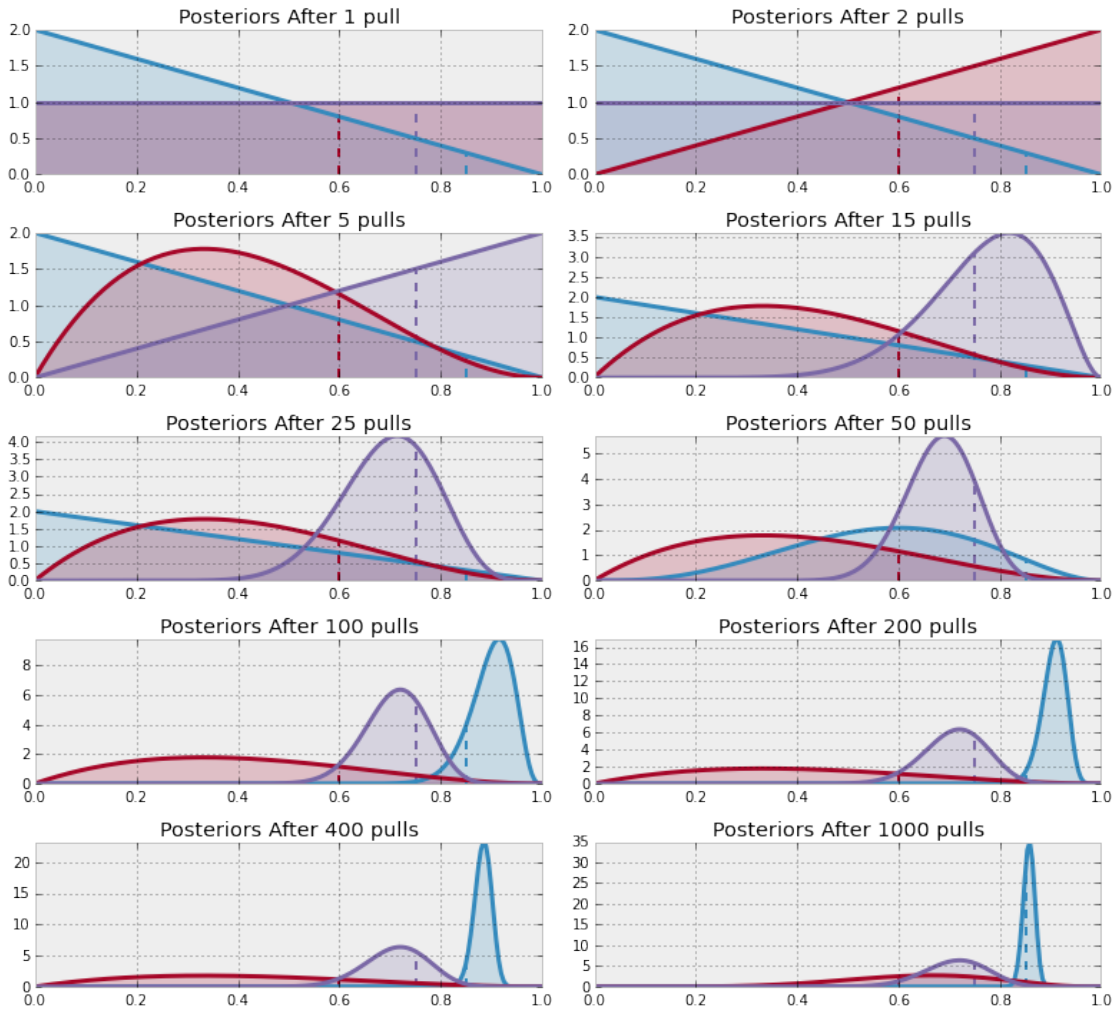
```
        sample_bandits(n): sample and train on n pulls.

    attributes:
        N: the cumulative number of samples
        choices: the historical choices as a (N,) array
        bb_score: the historical score as a (N,) array
    """

    def __init__(self, bandits):

        self.bandits = bandits
        n_bandits = len(self.bandits)
        self.wins = np.zeros(n_bandits)
        self.trials = np.zeros(n_bandits)
        self.N = 0
        self.choices = []
        self.bb_score = []

    def sample_bandits(self, n=1):

        bb_score = np.zeros(n)
        choices = np.zeros(n)

        for k in range(n):
            # sample from the bandits's priors, and select the largest sample
            choice = np.argmax(rbeta(1 + self.wins, 1 + self.trials - self.wins))

            # sample the chosen bandit
            result = self.bandits.pull(choice)

            # update priors and score
            self.wins[choice] += result
            self.trials[choice] += 1
            bb_score[k] = result
            self.N += 1
            choices[k] = choice

        self.bb_score = np.r_[self.bb_score, bb_score]
        self.choices = np.r_[self.choices, choices]
        return
```

Below we visualize the learning of the Bayesian Bandit solution.

In [6]:
```
figsize(11.0, 10)

beta = stats.beta
x = np.linspace(0.001, .999, 200)
```

```
        def plot_priors(bayesian_strategy, prob, lw=3, alpha=0.2, plt_vlines=True):
            # plotting function
            wins = bayesian_strategy.wins
            trials = bayesian_strategy.trials
            for i in range(prob.shape[0]):
                y = beta(1 + wins[i], 1 + trials[i] - wins[i])
                p = plt.plot(x, y.pdf(x), lw=lw)
                c = p[0].get_markeredgecolor()
                plt.fill_between(x, y.pdf(x), 0, color=c, alpha=alpha,
                                 label="underlying probability: %.2f" % prob[i])
                if plt_vlines:
                    plt.vlines(prob[i], 0, y.pdf(prob[i]),
                               colors=c, linestyles="--", lw=2)
                plt.autoscale(tight="True")
                plt.title("Posteriors After %d pull" % bayesian_strategy.N +
                          "s" * (bayesian_strategy.N > 1))
                plt.autoscale(tight=True)
            return
```

```
In [7]: hidden_prob = np.array([0.85, 0.60, 0.75])
        bandits = Bandits(hidden_prob)
        bayesian_strat = BayesianStrategy(bandits)

        draw_samples = [1, 1, 3, 10, 10, 25, 50, 100, 200, 600]

        for j, i in enumerate(draw_samples):
            plt.subplot(5, 2, j + 1)
            bayesian_strat.sample_bandits(i)
            plot_priors(bayesian_strat, hidden_prob)
            # plt.legend()
            plt.autoscale(tight=True)
        plt.tight_layout()
```



Note that we don't really care how accurate we become about the inference of the hidden probabilities — for this problem we are more interested in choosing the best bandit (or more accurately, becoming *more confident* in choosing the best bandit). For this reason, the distribution of the red bandit is very wide (representing ignorance about what that hidden probability might be) but we are reasonably confident that it is not the best, so the algorithm chooses to ignore it.

From the above, we can see that after 1000 pulls, the majority of the "blue" function leads the pack, hence we will almost always choose this arm. This is good, as this arm is indeed the best.

Below is a D3 app that demonstrates our algorithm updating/learning three bandits. The first figure shows the raw counts of pulls and wins, and the second figure is a dynamically updating plot. I encourage you to try to guess which bandit is optimal, prior to revealing the true probabilities, by selecting the `arm buttons`.

```
In [8]: from IPython.core.display import HTML

        # try executing the below command twice if the first time doesn't work
        HTML(filename="BanditsD3.html")
```

```
Out[8]: <IPython.core.display.HTML at 0x106822050>
```

Deviations of the observed ratio from the highest probability is a measure of performance. For example,in the long run, optimally we can attain the reward/pull ratio of the maximum bandit probability. Long-term realized ratios less than the maximum represent inefficiencies. (Realized ratios larger than the maximum probability is due to randomness, and will eventually fall below).

1.2.6 A Measure of *Good*

We need a metric to calculate how well we are doing. Recall the absolute *best* we can do is to always pick the bandit with the largest probability of winning. Denote this best bandit's probability of w_{opt}. Our score should be relative to how well we would have done had we chosen the best bandit from the beginning. This motivates the *total regret* of a strategy, defined as:

$$R_T = \sum_{i=1}^{T} \left(w_{opt} - w_{B(i)} \right) \tag{1}$$

$$\tag{2}$$

$$= Tw^* - \sum_{i=1}^{T} w_{B(i)} \tag{3}$$

where $w_{B(i)}$ is the probability of a prize of the chosen bandit in the ith round. A total regret of 0 means the strategy is attaining the best possible score. This is likely not possible, as initially our algorithm will often make the wrong choice. Ideally, a strategy's total regret should flatten as it learns the best bandit. (Mathematically, we achieve $w_{B(i)} = w_{opt}$ often)

Below we plot the total regret of this simulation, including the scores of some other strategies:

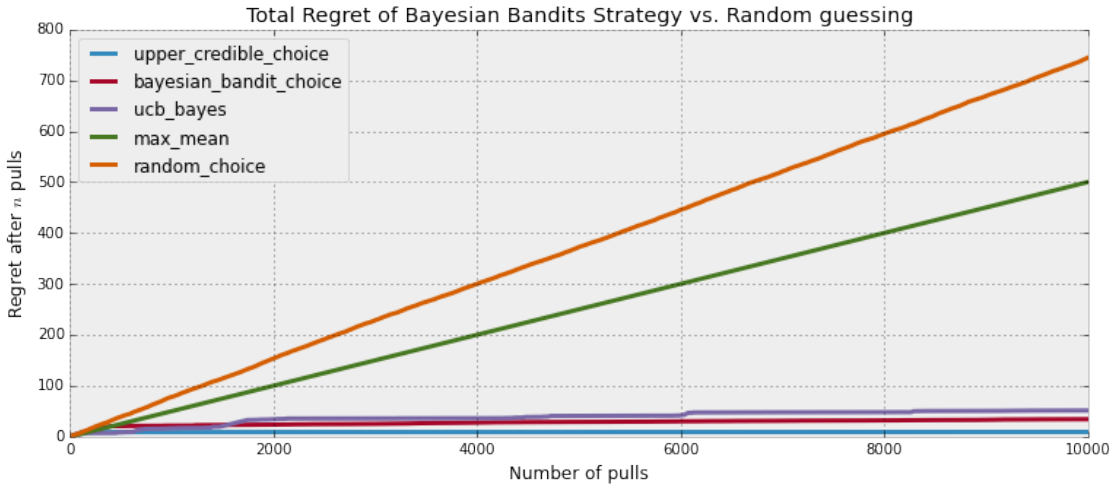
1. Random: randomly choose a bandit to pull. If you can't beat this, just stop.
2. Largest Bayesian credible bound: pick the bandit with the largest upper bound in its 95% credible region of the underlying probability.
3. Bayes-UCB algorithm: pick the bandit with the largest *score*, where score is a dynamic quantile of the posterior (see [4])
4. Mean of posterior: choose the bandit with the largest posterior mean. This is what a human player (sans computer) would likely do.
5. Largest proportion: pick the bandit with the current largest observed proportion of winning.

The code for these are in the `other_strats.py`, where you can implement your own strategy very easily.

```
In [9]: figsize(12.5, 5)
        from other_strats import GeneralBanditStrat, bayesian_bandit_choice, max_mean, lower_c
                                 upper_credible_choice, random_choice, ucb_bayes, Bandits

        # define a harder problem
        hidden_prob = np.array([0.15, 0.2, 0.1, 0.05])
        bandits = Bandits(hidden_prob)

        # define regret
```

```
    def regret(probabilities, choices):
        w_opt = probabilities.max()
        return (w_opt - probabilities[choices.astype(int)]).cumsum()

    # create new strategies
    strategies = [upper_credible_choice,
                  bayesian_bandit_choice,
                  ucb_bayes,
                  max_mean,
                  random_choice]
    algos = []
    for strat in strategies:
        algos.append(GeneralBanditStrat(bandits, strat))
```

```
In [10]: # train 10000 times
         for strat in algos:
             strat.sample_bandits(10000)

         #test and plot
         for i, strat in enumerate(algos):
             _regret = regret(hidden_prob, strat.choices)
             plt.plot(_regret, label=strategies[i].__name__, lw=3)

         plt.title("Total Regret of Bayesian Bandits Strategy vs. Random guessing")
         plt.xlabel("Number of pulls")
         plt.ylabel("Regret after $n$ pulls");
         plt.legend(loc="upper left");
```



Like we wanted, Bayesian bandits and other strategies have decreasing rates of regret, representing that we are achieving optimal choices. To be more scientific so as to remove any possible luck in the above simulation, we should instead look at the *expected total regret*:

$$\bar{R_T} = E[R_T]$$

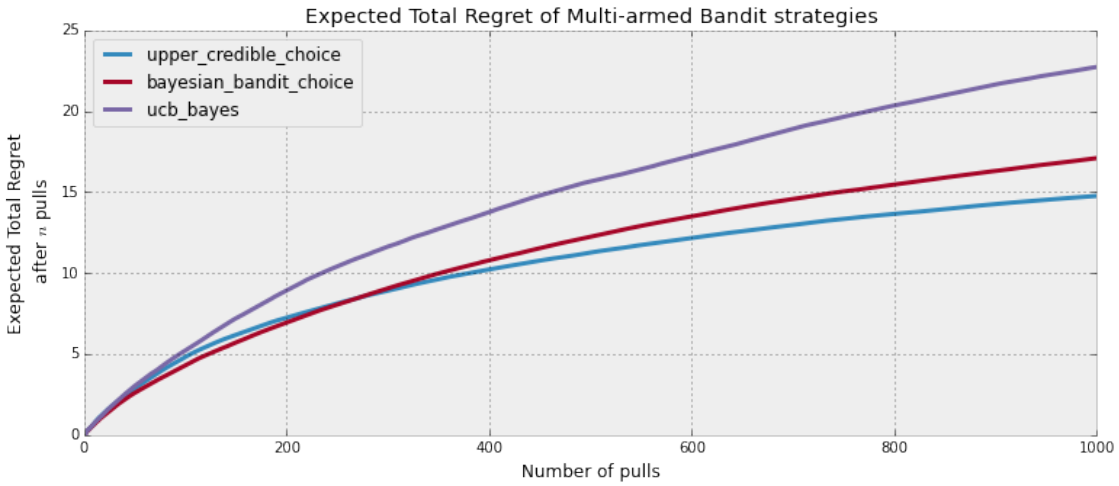
It can be shown that any *sub-optimal* strategy's expected total regret is bounded below logarithmically. Formally:

$$E[R_T] = \Omega\left(\log(T)\right)$$

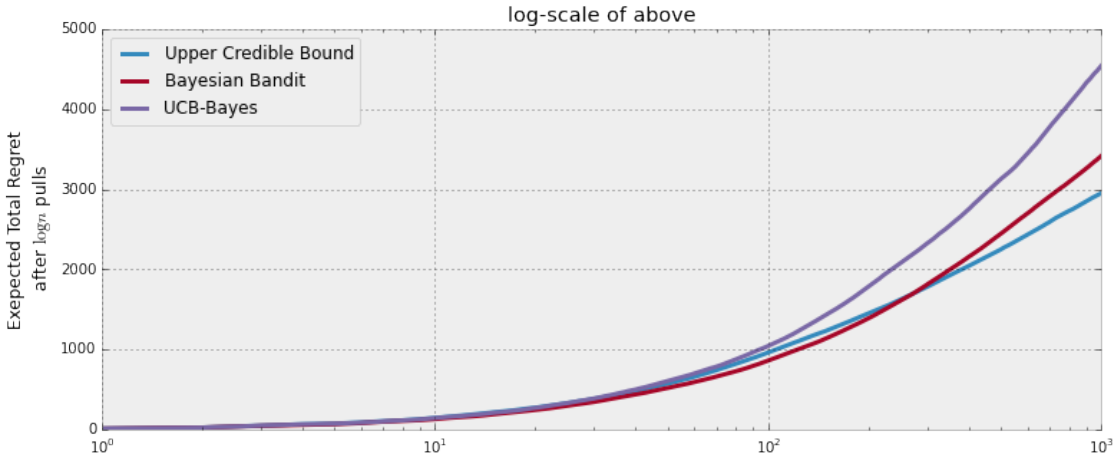
Thus, any strategy that matches logarithmic-growing regret is said to "solve" the Multi-Armed Bandit problem [3].

Using the Law of Large Numbers, we can approximate Bayesian Bandit's expected total regret by performing the same experiment many times (500 times, to be fair):

```
In [14]: # this can be slow, so I recommend NOT running it.
         trials = 200
         expected_total_regret = np.zeros((1000, 3))

         for i_strat, strat in enumerate(strategies[:-2]):
             for i in range(trials):
                 general_strat = GeneralBanditStrat(bandits, strat)
                 general_strat.sample_bandits(1000)
                 _regret = regret(hidden_prob, general_strat.choices)
                 expected_total_regret[:, i_strat] += _regret

             plt.plot(expected_total_regret[:, i_strat] / trials, lw=3, label=strat.__name__)

         plt.title("Expected Total Regret of Multi-armed Bandit strategies")
         plt.xlabel("Number of pulls")
         plt.ylabel("Exepected Total Regret \n after $n$ pulls");
         plt.legend(loc="upper left");
```



```
In [15]: plt.figure()
         [pl1, pl2, pl3] = plt.plot(expected_total_regret[:, [0, 1, 2]], lw=3)
         plt.xscale("log")
         plt.legend([pl1, pl2, pl3],
                    ["Upper Credible Bound", "Bayesian Bandit", "UCB-Bayes"],
                    loc="upper left")
         plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
         plt.title("log-scale of above");
         plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
```



1.2.7 Extending the algorithm

Because of the Bayesian Bandits algorithm's simplicity, it is easy to extend. Some possibilities are:

- If interested in the *minimum* probability (eg: where prizes are a bad thing), simply choose $B = \text{argmin } X_b$ and proceed.
- Adding learning rates: Suppose the underlying environment may change over time. Technically the standard Bayesian Bandit algorithm would self-update itself (awesome) by noting that what it thought was the best is starting to fail more often. We can motivate the algorithm to learn changing environments quicker by simply adding a *rate* term upon updating:

```
self.wins[ choice ] = rate*self.wins[ choice ] + result
self.trials[ choice ] = rate*self.trials[ choice ] + 1
```

If `rate < 1`, the algorithm will *forget* its previous wins quicker and there will be a downward pressure towards ignorance. Conversely, setting `rate > 1` implies your algorithm will act more risky, and bet on earlier winners more often and be more resistant to changing environments.

- Hierarchical algorithms: We can setup a Bayesian Bandit algorithm on top of smaller bandit algorithms. Suppose we have N Bayesian Bandit models, each varying in some behavior

(for example different `rate` parameters, representing varying sensitivity to changing environments). On top of these N models is another Bayesian Bandit learner that will select a sub-Bayesian Bandit. This chosen Bayesian Bandit will then make an internal choice as to which machine to pull. The super-Bayesian Bandit updates itself depending on whether the sub-Bayesian Bandit was correct or not.

- Extending the rewards, denoted y_a for bandit a, to random variables from a distribution $f_{y_a}(y)$ is straightforward. More generally, this problem can be rephrased as "Find the bandit with the largest expected value", as playing the bandit with the largest expected value is optimal. In the case above, f_{y_a} was Bernoulli with probability p_a, hence the expected value for a bandit is equal to p_a, which is why it looks like we are aiming to maximize the probability of winning. If f is not Bernoulli, and it is non-negative, which can be accomplished a priori by shifting the distribution (we assume we know f), then the algorithm behaves as before:

For each round,

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result,$R \sim f_{y_b}$, of pulling bandit B, and update your prior on bandit B.
4. Return to 1

The issue is in the sampling of the X_b drawing phase. With Beta priors and Bernoulli observations, we have a Beta posterior — this is easy to sample from. But now, with arbitrary distributions f, we have a non-trivial posterior. Sampling from these can be difficult.

- There has been some interest in extending the Bayesian Bandit algorithm to commenting systems. Recall in Chapter 4, we developed a ranking algorithm based on the Bayesian lower-bound of the proportion of upvotes to the total number of votes. One problem with this approach is that it will bias the top rankings towards older comments, since older comments naturally have more votes (and hence the lower-bound is tighter to the true proportion). This creates a positive feedback cycle where older comments gain more votes, hence are displayed more often, hence gain more votes, etc. This pushes any new, potentially better comments, towards the bottom. J. Neufeld proposes a system to remedy this that uses a Bayesian Bandit solution.

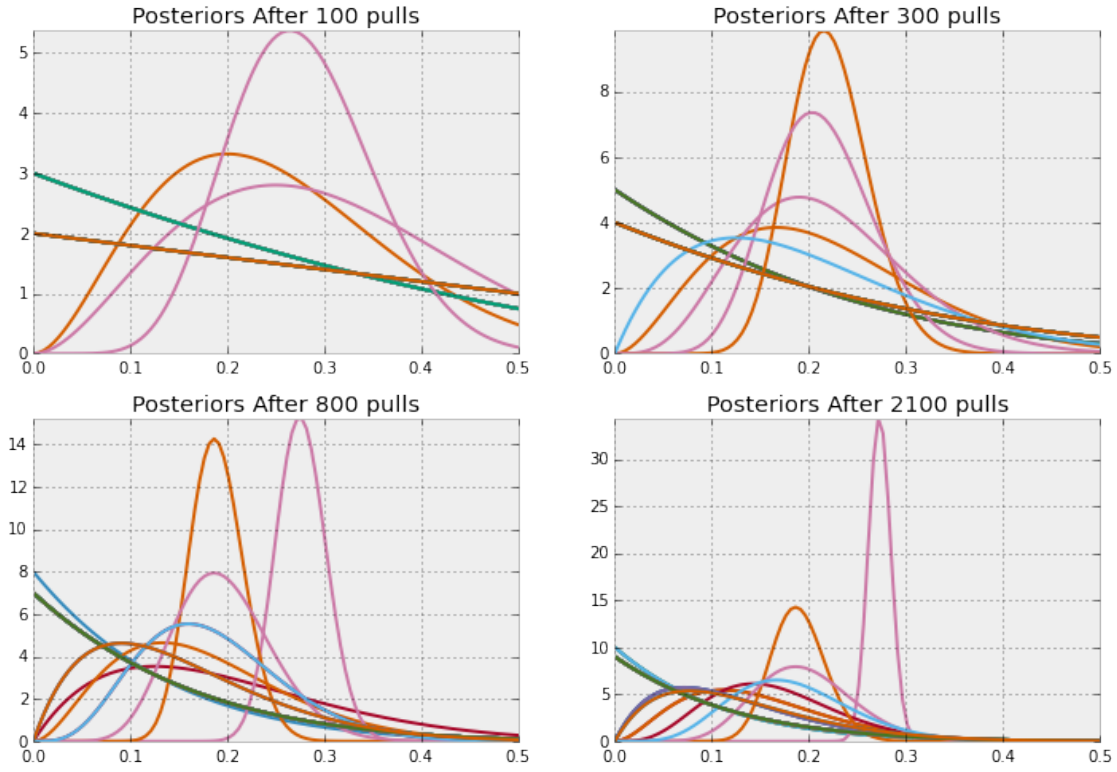
His proposal is to consider each comment as a Bandit, with the number of pulls equal to the number of votes cast, and number of rewards as the number of upvotes, hence creating a Beta$(1 + U, 1 + D)$ posterior. As visitors visit the page, samples are drawn from each bandit/comment, but instead of displaying the comment with the max sample, the comments are ranked according to the ranking of their respective samples. From J. Neufeld's blog [7]:

> [The] resulting ranking algorithm is quite straightforward, each new time the comments page is loaded, the score for each comment is sampled from a Beta$(1 + U, 1 + D)$, comments are then ranked by this score in descending order... This randomization has a unique benefit in that even untouched comments $(U = 0, D = 0)$ have some chance of being seen even in threads with 5000+ comments (something that is not happening now), but, at the same time, the user is not likely to be inundated with rating these new comments.

Just for fun, though the colors explode, we watch the Bayesian Bandit algorithm learn 35 different options.

```
In [16]: figsize(12.0, 8)
         beta = stats.beta
         hidden_prob = beta.rvs(1, 13, size=35)
         print(hidden_prob)
         bandits = Bandits(hidden_prob)
         bayesian_strat = BayesianStrategy(bandits)

         for j, i in enumerate([100, 200, 500, 1300]):
             plt.subplot(2, 2, j + 1)
             bayesian_strat.sample_bandits(i)
             plot_priors(bayesian_strat, hidden_prob, lw=2, alpha=0.0, plt_vlines=False)
             # plt.legend()
             plt.xlim(0, 0.5)
```

```
[ 0.0431  0.0745  0.1187  0.0098  0.0945  0.0438  0.0442  0.0059  0.0749
  0.023   0.0543  0.025   0.1231  0.0148  0.0164  0.2688  0.0073  0.0564
  0.0031  0.0698  0.0478  0.1657  0.0091  0.0384  0.2236  0.1548  0.0562
  0.0209  0.024   0.0197  0.0788  0.0572  0.1207  0.0405  0.0679]
```



1.3 Eliciting expert prior

Specifying a subjective prior is how practitioners incorporate domain knowledge about the problem into our mathematical framework. Allowing domain knowledge is useful for many reasons, for example:

- Aids the speed of MCMC convergence. For example, if we know the unknown parameter is strictly positive, then we can restrict our attention there, hence saving time that would otherwise be spent exploring negative values.
- More accurate inference. By weighing prior values near the true unknown value higher, we are narrowing our eventual inference (by making the posterior tighter around the unknown)
- Express our uncertainty better. See the *Price is Right* problem in Chapter 5.

Of course, practitioners of Bayesian methods are not experts in every field, so we must turn to domain experts to craft our priors. We must be careful with how we elicit these priors though. Some things to consider:

1. From experience, I would avoid introducing Betas, Gammas, etc. to non-Bayesian practitioners. Furthermore, non-statisticians can get tripped up by how a continuous probability function can have a value exceeding one.

2. Individuals often neglect the rare *tail-events* and put too much weight around the mean of distribution.

3. Related to above is that almost always individuals will under-emphasize the uncertainty in their guesses.

Eliciting priors from non-technical experts is especially difficult. Rather than introduce the notion of probability distributions, priors, etc. that may scare an expert, there is a much simpler solution.

1.3.1 Trial roulette method

The *trial roulette method* [8] focuses on building a prior distribution by placing counters (think casino chips) on what the expert thinks are possible outcomes. The expert is given N counters (say $N = 20$) and is asked to place them on a pre-printed grid, with bins representing intervals. Each column would represent their belief of the probability of getting the corresponding bin result. Each chip would represent an $\frac{1}{N} = 0.05$ increase in the probability of the outcome being in that interval. For example [9]:

> A student is asked to predict the mark in a future exam. The figure below shows a completed grid for the elicitation of a subjective probability distribution. The horizontal axis of the grid shows the possible bins (or mark intervals) that the student was asked to consider. The numbers in top row record the number of chips per bin. The completed grid (using a total of 20 chips) shows that the student believes there is a 30% chance that the mark will be between 60 and 64.9.

From this, we can fit a distribution that captures the expert's choice. Some reasons in favor of using this technique are: