9
Bayesian Machine Learning

In this chapter, we will introduce Bayesian approaches to machine learning, and how their different perspectives on uncertainty add value when developing and evaluating algorithmic trading strategies.

Bayesian statistics allow us to quantify the uncertainty about future events and refine our estimates in a principled way as new information arrives. This dynamic approach adapts well to the evolving nature of financial markets. It is particularly useful when there is less relevant data and we require methods that systematically integrate prior knowledge or assumptions.

We will see that Bayesian approaches to machine learning allow for richer insights into the uncertainty around statistical metrics, parameter estimates, and predictions. The applications range from more granular risk management to dynamic updates of predictive models that incorporate changes in the market environment. The Black-Litterman approach to asset allocation (see `Chapter 5`, *Strategy Evaluation*, can be interpreted as a Bayesian model. It computes the expected return as an average of the market equilibrium and the investor's views, weighted by each asset's volatility, cross-asset correlations, and the confidence in each forecast.

More specifically, in this chapter, we will cover the following topics:

- How Bayesian statistics apply to machine learning
- How to use probabilistic programming with PyMC3
- How to define and train machine learning models
- How to run state-of-the-art sampling methods to conduct approximate inference
- How to apply Bayesian machine learning to compute dynamic Sharpe ratios, build Bayesian classifiers, and estimate stochastic volatility

References, links to additional material, and the code examples for this chapter are in the corresponding directory of the GitHub repository. Please follow the installation instructions provided in `Chapter 1`, *Machine Learning for Trading*.

How Bayesian machine learning works

Classical statistics is also called frequentist because it interprets probability as the relative frequency of an event over the long run, that is, after observing a large number of trials. In the context of probabilities, an event is a combination of one or more elementary outcomes of an experiment, such as any of six equal results in rolls of two dice or an asset price dropping by 10% or more on a given day.

Bayesian statistics, in contrast, views probability as a measure of the confidence or belief in the occurrence of an event. The Bayesian perspective of probability leaves more room for subjective views and, consequently, differences in opinions than the frequentist interpretation. This difference is most striking for events that do not happen often enough to arrive at an objective measure of long-term frequency.

Put differently, frequentist statistics assume that data is a random sample from a population and aims to identify the fixed parameters that generated the data. Bayesian statistics, in turn, take the data as given and considers the parameters to be random variables with a distribution that can be inferred from data. As a result, frequentist approaches require at least as many data points as there are parameters to be estimated. Bayesian approaches, on the other hand, are compatible with smaller datasets and are well-suited for online learning, one sample at a time.

The Bayesian view is very useful for many real-world events that are rare or unique, at least in important respects. Examples include the outcome of the next election or the question of whether the markets will crash within three months. In each case, there is both relevant historical data as well as unique circumstances that unfold as the event approaches.

First, we will introduce Bayes' theorem, which crystallizes the concept of updating beliefs by combining prior assumptions with new empirical evidence and comparing the resulting parameter estimates with their frequentist counterparts. We will then demonstrate two approaches to Bayesian statistical inference that produce insights into the posterior distribution of the latent, that is, unobserved parameters, such as their expected values, under different circumstances:

1. Conjugate priors facilitate the updating process by providing a closed-form solution, but exact, analytical methods are not always available.
2. Approximate inference simulates the distribution that results from combining assumptions and data and uses samples from this distribution to compute statistical insights.

How to update assumptions from empirical evidence

The theorem that Reverend Thomas Bayes came up with over 250 years ago uses fundamental probability theory to prescribe how probabilities or beliefs should change as relevant new information arrives. The following quote by – John Maynard Keynes captures the Bayesian mindset:

> *"When the facts change, I change my mind. What do you do, sir?"*

It relies on the conditional and total probability and the chain rule; see the references on GitHub for reviews of these concepts.

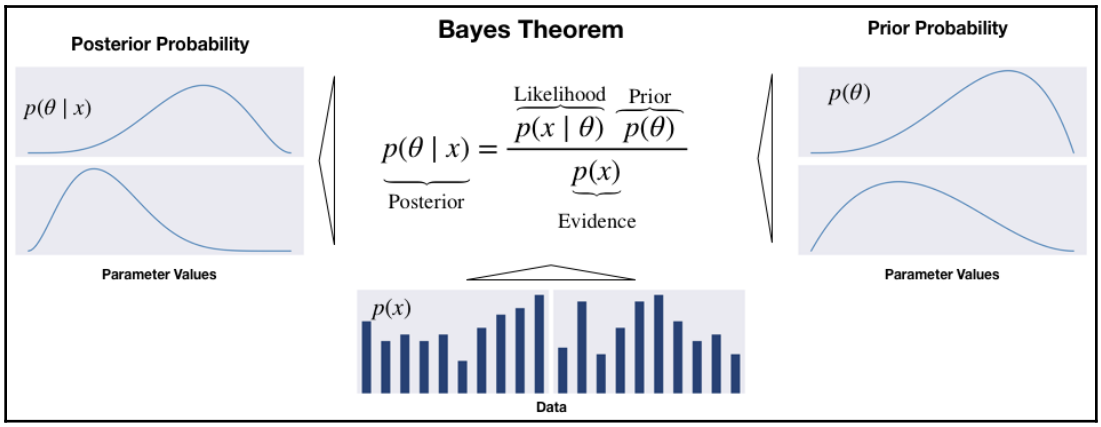
The belief concerns a single or vector of parameters θ (also called hypotheses). Each parameter can be discrete or continuous. θ could be a one-dimensional statistic like the (discrete) mode of a categorical variable or a (continuous) mean, or a higher dimensional set of values like a covariance matrix or the weights of a deep neural network.

A key difference of frequentist statistics is that Bayesian assumptions are expressed as probability distributions rather than parameter values. Consequently, while frequentist inference focuses on point estimates, Bayesian inference yields probability distributions.

Bayes' Theorem updates the beliefs about the parameters of interest by computing the posterior probability distribution from the following inputs, as shown in the following diagram:

- The **prior** distribution indicates how likely we consider each possible hypothesis.
- The **likelihood function** outputs the probability of observing a dataset given certain values for the θ parameters.

- The **evidence** measures how likely the observed data is given all possible hypotheses. Hence, it is the same for all parameter values and serves to normalize the numerator:



Bayes Theorem

The posterior is the product of prior and likelihood, divided by the evidence, and reflects the updated probability distribution of the hypotheses, taking into account both prior assumptions and the data. Viewed differently, the product of the prior and the likelihood results from applying the chain rule to factorize the joint distribution of data and parameters.

With higher-dimensional, continuous variables, the formulation becomes more complex and involves (multiple) integrals. An alternative formulation uses odds to express the posterior odds as the product of the prior odds times the likelihood ratio (see the references for more details).

Exact inference: Maximum a Posteriori estimation

Practical applications of Bayes' rule to exactly compute posterior probabilities are quite limited because the computation of the evidence term in the denominator is quite challenging. The evidence reflects the probability of the observed data over all possible parameter values. It is also called the marginal likelihood because it requires *marginalizing out* the parameters' distribution by adding or integrating over their distribution. This is generally only possible in simple cases with a small number of discrete parameters that assume very few values.

Maximum a posteriori probability (MAP) estimation leverages that the evidence is a constant factor that scales the posterior to meet the requirements for a probability distribution. Since the evidence does not depend on θ, the posterior distribution is proportional to the product of the likelihood and the prior. Hence, MAP estimation chooses the value of θ that maximizes the posterior given the observed data and the prior belief, that is, the mode of the posterior.

The MAP approach contrasts with the **maximum likelihood estimation** (**MLE**) of parameters, which define a probability distribution. MLE picks the parameter value θ that maximizes the likelihood function for the observed training data.

A look at the definitions highlights that MAP differs from MLE by including the prior distribution. In other words, unless the prior is a constant, the MAP estimate θ will differ from its MLE counterpart:

$$\theta_{MLE} = arg\,\max_{\theta} P(X|\theta)$$

$$\theta_{MAP} = arg\,\max_{\theta} P(X|\theta)P(\theta)$$

The MLE solution tends to reflect the frequentist notion that probability estimates should reflect observed ratios. On the other hand, the impact of the prior on the MAP estimate often corresponds to adding data that reflects the prior assumptions to the MLE. For example, a strong prior that a coin is biased can be incorporated in the MLE context by adding skewed trial data.

Prior distributions are a critical ingredient for Bayesian models. We will now introduce some convenient choices that facilitate analytical inference.

How to select priors

The prior should reflect knowledge of the distribution of the parameters because it influences the MAP estimate. If a prior is not known with certainty, we need to make a choice, often from several reasonable options. In general, it is good practice to justify the prior and check for robustness by testing whether alternatives lead to the same conclusion.

There are several types of priors:

- **Objective** priors maximize the impact of the data on the posterior. If the parameter distribution is unknown, we can select an uninformative prior like a uniform distribution, also called a flat prior, over a relevant range of parameter values.

- In contrast, **subjective** priors aim to incorporate information that's external to the model into the estimate.
- An **empirical** prior combines Bayesian and frequentist methods and uses historical data to eliminate subjectivity, such as by estimating various moments to fit a standard distribution.

In the context of a machine learning model, the prior can be viewed as a regularizer because it limits the values that the posterior can assume. Parameters that have zero prior probability, for example, are not part of the posterior distribution. Generally, more good data allows for stronger conclusions and reduces the influence of the prior.

How to keep inference simple – conjugate priors

A prior distribution is conjugate with respect to the likelihood when the resulting posterior is of the same type of distribution as the prior, except for different parameters. When both the prior and the likelihood are normally distributed, then the posterior is also normally distributed.

The conjugacy of the prior and likelihood implies a closed-form solution for the posterior that facilitates the update process and avoids the need to use numerical methods to approximate the posterior. Moreover, the resulting posterior can be used as prior for the next update step.

Let's illustrate this process using a binary classification example for stock price movements.

How to dynamically estimate the probabilities of asset price moves

When the data consists of binary Bernoulli random variables with a certain success probability for a positive outcome, the number of successes in repeated trials follows a Binomial distribution. The conjugate prior is the Beta distribution with support over the interval [0, 1] and two shape parameters to model arbitrary prior distributions over the success probability. Hence, the posterior distribution is also a Beta distribution that we can derive by directly updating the parameters.

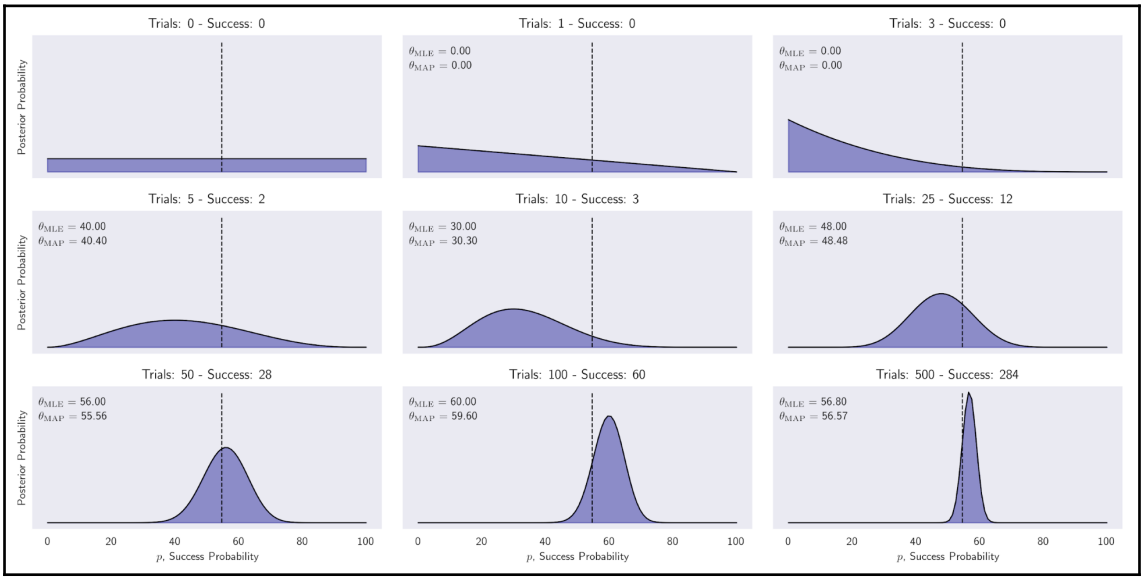
We will collect samples of different sizes of binarized daily S&P 500 returns, where the positive outcome is a price increase. Starting from an uninformative prior that allocates equal probability to each possible success probability in the interval [0, 1], we compute the posterior for different evidence samples.

The following code sample shows that the update consists of simply adding the observed numbers of success and failure to the parameters of the prior distribution to obtain the posterior:

```
n_days = [0, 1, 3, 5, 10, 25, 50, 100, 500]
outcomes = sp500_binary.sample(n_days[-1])
p = np.linspace(0, 1, 100)

# uniform (uninformative) prior
a = b = 1
for i, days in enumerate(n_days):
    up = outcomes.iloc[:days].sum()
    down = days - up
    update = stats.beta.pdf(p, a + up , b + down)
```

The resulting posterior distributions are plotted in the following graphs. They illustrate the evolution from a uniform prior that views all success probabilities as equally likely to an increasingly peaked distribution.

After 500 samples, the probability is concentrated near the actual probability of a positive move at 54.7% from 2010 to 2017. It also shows the small differences between MLE and MAP estimates, where the latter tends to be pulled slightly toward the expected value of the uniform prior, as shown in the following diagram:



Posterior probabilities

In practice, the use of conjugate priors is limited to low-dimensional cases. In addition, the simplified MAP approach avoids computing the evidence term, but has several shortcomings even when it is available; it does not return a distribution so that we can derive a measure of uncertainty, or use it as a prior. Hence, we need to resort to approximates rather than exact inference using numerical methods and stochastic simulation, which we will introduce next.

Approximate inference: stochastic versus deterministic approaches

For most models of practical relevance, it will not be possible to derive the exact posterior distribution analytically and compute the expected values for the latent parameters. The model may have too many parameters, or the posterior distribution may be too complex for an analytical solution. For continuous variables, the integrals may not have closed-form solutions, while the dimensionality of the space and the complexity of the integrand may prohibit numerical integration. For discrete variables, the marginalizations involve summing over all possible configurations of the hidden variables, and though this is always possible in principle, we often find in practice that there may be exponentially many hidden states so that exact calculation is prohibitively expensive.

Although for some applications the posterior distribution over unobserved parameters will be of interest, more often than not it is primarily required to evaluate expectations, for example, to make predictions. In such situations, we can rely on approximate inference:

- **Stochastic** techniques based on **Markov Chain Monte Carlo (MCMC)** sampling have popularized the use of Bayesian methods across many domains. They generally have the ability to converge to the exact result. In practice, sampling methods can be computationally demanding and are often limited to small-scale problems.
- **Deterministic** methods, known as variational inference or variational Bayes, are based on analytical approximations to the posterior distribution and can scale well to large applications. They make simplified assumptions, for example, that the posterior factorizes in a particular way or it has a specific parametric form such as a Gaussian. Hence, they do not generate exact results and can be used as complements to sampling methods.

Sampling-based stochastic inference

Sampling is about drawing samples, $X=(x_1, ..., x_n)$, from a given distribution, $p(x)$. Assuming the samples are independent, the law of large numbers ensures that for a growing number of samples, the fraction of a given instance, x_i, in the sample (for the discrete case) corresponds to its probability, $p(x=x_i)$. In the continuous case, the analogous reasoning applies to a given region of the sample space. Hence, averages over samples can be used as unbiased estimators of the expected values of parameters of the distribution.

A practical challenge consists in ensuring independent sampling because the distribution is unknown. Dependent samples may still be unbiased, but tend to increase the variance of the estimate so that more samples will be needed for an equally precise estimate as for independent samples.

Sampling from a multivariate distribution is computationally demanding as the number of states increases exponentially with the number of dimensions. Numerous algorithms facilitate the process (see references for an overview). Now, we will introduce a few popular variations of MCMC-based methods.

Markov chain Monte Carlo sampling

A Markov chain is a dynamic stochastic model that describes a random walk over a set of states, connected by transition probabilities. The Markov property stipulates that the process has no memory, and the next step only depends on the current state. In other words, it's conditional on the present, past, and future being independent, that is, information about past states does not help to predict the future beyond what we know from the present.

Monte Carlo methods rely on repeated random sampling to approximate results that may be deterministic, but that does not permit an analytic, exact solution. It was developed during the Manhattan Project to estimate energy at the atomic level and received its enduring code name to ensure secrecy.

Many algorithms apply the Monte Carlo method to a Markov Chain, and generally proceed as follows:

1. Start at the current position.
2. Draw a new position from a proposal distribution.

3. Evaluate the probability of the new position in light of data and prior distributions:
    1. If sufficiently likely, move to the new position
    2. Otherwise, remain at the current position
4. Repeat from step 1.
5. After a given number of iterations, return all accepted positions.

MCMC aims to identify and explore interesting regions of the posterior that concentrate on significant probability density. The memoryless process is said to converge when it consistently moves through nearby high probability states of the posterior where the acceptance rate increases. A key challenge is to balance the need for random exploration of the sample space with the risk of reducing the acceptance rate.

The initial steps of this process are likely to be more reflective of the starting position than the posterior and are typically discarded as **burn-in** samples. A key MCMC property is that the process should forget about its initial position after a certain (but unknown) number of iterations.

The remaining samples are called the trace of the process. Assuming convergence, the relative frequency of samples approximates the posterior and can be used to compute expected values based on the law of large numbers.

As indicated previously, the precision of the estimate depends on the serial correlation of the samples collected by the random walk, each of which, by design, depends only on the previous state. Higher correlation limits the effective exploration of the posterior and needs to be subjected to diagnostic tests.

General techniques to design such a Markov chain include Gibbs sampling, the Metropolis-Hastings algorithm, and more recent Hamiltonian MCMC methods that tend to perform better.

Gibbs sampling

Gibbs sampling simplifies multivariate sampling to a sequence of one-dimensional draws. From a starting point, it iteratively holds n-1 variables constant while sampling the n^{th} variable. It incorporates this sample and repeats.

The algorithm is very simple and easy to implement but produces highly correlated samples that slow down convergence. Its sequential nature also prevents parallelization.

Metropolis-Hastings sampling

The Metropolis-Hastings algorithm randomly proposes new locations based on its current state to effectively explore the sample space and reduce the correlation of samples relative to Gibbs sampling. To ensure that it samples from the posterior, it evaluates the proposal using the product of prior and likelihood, which is proportional to the posterior. It accepts with a probability that depends on the result, which is relative to the corresponding value for the current sample.

A key benefit of the proposal evaluation method is that it works with a proportional evaluation rather than an exact evaluation of the posterior. However, it can take a long time to converge because the random movements that are not related to the posterior can reduce the acceptance rate so that a large number of steps produces only a small number of (potentially correlated) samples. The acceptance rate can be tuned by reducing the variance of the proposal distribution, but the resulting smaller steps imply less exploration.

Hamiltonian Monte Carlo – going NUTS

Hamiltonian Monte Carlo (HMC) is a hybrid method that leverages the first-order derivative information of the gradient of the likelihood to propose new states for exploration and overcome some of the challenges of MCMC. In addition, it incorporates momentum to efficiently jump around the posterior. As a result, it converges faster to a high-dimensional target distribution than simpler random-walk Metropolis or Gibbs sampling.

The No-U-Turn sampler is a self-tuning HMC extension that adaptively regulates the size and number of moves around the posterior before selecting a proposal. It works well on high-dimensional and complex posterior distributions and allows many complex models to be fit without specialized knowledge about the fitting algorithm itself. As we will see in the next section, it is the default sampler in PyMC3.

Variational Inference

Variational Inference (VI) is a machine learning method that approximates probability densities through optimization. In the Bayesian context, it approximates the posterior distribution as follows:

1. Select a parametrized family of probability distributions
2. Find the member of this family closest to the target, as measured by Kullback-Leibler divergence

Compared to MCMC, Variational Bayes tends to converge faster and scales to large data better. While MCMC approximates the posterior with samples from the chain that will eventually converge arbitrarily close to the target, variational algorithms approximate the posterior with the result of the optimization, which is not guaranteed to coincide with the target.

Variational Inference is better suited for large datasets and to quickly explore many models. In contrast, MCMC will deliver more accurate results on smaller datasets or when time and computational resources pose fewer constraints.

Automatic Differentiation Variational Inference (ADVI)

The downside of Variational Inference is the need for model-specific derivations and the implementation of a tailored optimization routine that has slowed down widespread adoption.

The recent **Automatic Differentiation Variational Inference (ADVI)** algorithm automates this process so that the user only specifies the model, expressed as a program, and ADVI automatically generates a corresponding variational algorithm (see references on GitHub for implementation details).

We will see that PyMC3 supports various Variational Inference techniques, including ADVI.

Probabilistic programming with PyMC3

Probabilistic programming provides a language to describe and fit probability distributions so that we can design, encode, and automatically estimate and evaluate complex models. It aims to abstract away some of the computational and analytical complexity to allow us to focus on the conceptually more straightforward and intuitive aspects of Bayesian reasoning and inference.

The field has become quite dynamic since new languages emerged. Uber open sourced Pyro (based on PyTorch) and Google recently added a probability module to TensorFlow (see the resources linked on GitHub).

As a result, the practical relevance and use of Bayesian methods in machine learning will likely increase to generate insights into uncertainty and for use cases that require transparent rather than black-box models in particular.

In this section, we will introduce the popular PyMC3 library, which implements advanced MCMC sampling and Variational Inference for machine learning models using Python. Together with Stan, named after Stanislaw Ulam, who invented the Monte Carlo method, and developed by Andrew Gelman at Columbia University since 2012, it is the most popular probabilistic programming language.

Bayesian machine learning with Theano

PyMC3 was released in January 2017 to add Hamiltonian MC methods to the Metropolis-Hastings sampler that's used in PyMC2 (released in 2012). PyMC3 uses Theano as its computational backend for dynamic C compilation and automatic differentiation. Theano is a matrix-focused and GPU-enabled optimization library that was developed at Yoshua Bengio's Montreal Institute for Learning Algorithms (MILA) and inspired TensorFlow. MILA recently ceased to further develop Theano due to the success of newer deep learning libraries (see Chapter 16 *Deep Learning* for details). PyMC4, which is planned for 2019, will use TensorFlow instead, with presumably limited impact on the API.

The PyMC3 workflow

PyMC3 aims for intuitive and readable, yet powerful syntax that reflects how statisticians describe models. The modeling process generally follows these five steps:

1. Encode a probability model by defining the following:
    1. The prior distributions that quantify knowledge and uncertainty about latent variables
    2. The likelihood function that conditions the parameters on observed data
2. Analyze the posterior using one of the options described in the previous section:
    1. Obtain a point estimate using MAP inference
    2. Sample from the posterior using MCMC methods
3. Approximate the posterior using variational Bayes.
4. Check your model using various diagnostic tools.
5. Generate predictions.

The resulting model can be used for inference to gain detailed insights into parameter values as well as to predict outcomes for new data points.

We will illustrate this workflow using simple logistic regression (see the notebook bayesian_logistic_regression). Subsequently, we will use PyMC3 to compute and compare Bayesian Sharpe ratios, estimate dynamic pairs trading ratios, and implement Bayesian linear time series models.

Model definition – Bayesian logistic regression

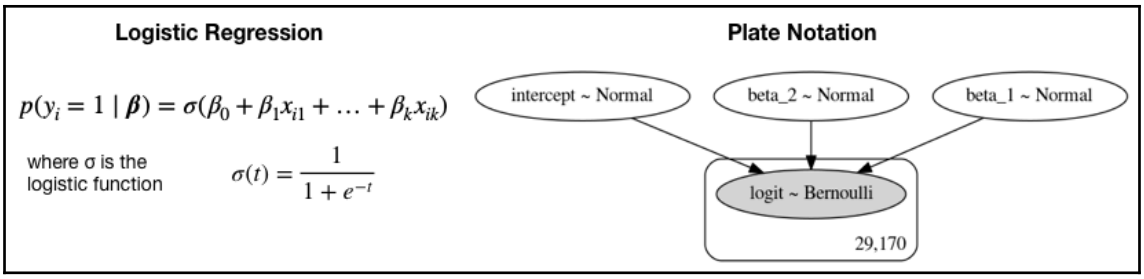
As discussed in `Chapter 6`, *Machine Learning Workflow*, logistic regression estimates a linear relationship between a set of features and a binary outcome, which is mediated by a sigmoid function to ensure that the model produces probabilities. The frequentist approach resulted in point estimates for the parameters that measure the influence of each feature on the probability that a data point belongs to the positive class, with confidence intervals based on assumptions about the parameter distribution.

In contrast, Bayesian logistic regression estimates the posterior distribution over the parameters itself. The posterior allows for more robust estimates of what is called a Bayesian credible interval for each parameter, with the benefit of more transparency about the model's uncertainty.

A probabilistic program consists of observed and unobserved random variables (RVs). As we have discussed, we define the observed RVs via likelihood distributions and unobserved RVs via prior distributions. PyMC3 includes numerous probability distributions for this purpose.

We will use a simple dataset that classifies 30,000 individuals by income using a threshold of $50K per year. This dataset will contain information on age, sex, hours worked, and years of education. Hence, we are modeling the probability that an individual earns more than $50K using these features.

The PyMC3 library makes it very straightforward to perform approximate Bayesian inference for logistic regression. Logistic regression models the probability that individual *i* earns a high income based on *k* features, as outlined on the left-hand side of the following diagram:



We will use the context manager `with` to define a `manual_logistic_model` that we can refer to later as a probabilistic model:

1. The random variables for the unobserved parameters for intercept and two features are expressed using uninformative priors that assume normal distributions with a mean of 0 and a standard deviation of 100.
2. The likelihood combines the parameters with the data according to the specification of the logistic regression.
3. The outcome is modeled as a Bernoulli RV with success probability given by the likelihood:

```
with pm.Model() as manual_logistic_model:
    # coefficients as rvs with uninformative priors
    intercept = pm.Normal('intercept', 0, sd=100)
    b1 = pm.Normal('beta_1', 0, sd=100)
    b2 = pm.Normal('beta_2', 0, sd=100)

    # Likelihood transforms rvs into probabilities p(y=1)
    # according to logistic regression model.
    likelihood = pm.invlogit(intercept + b1 * data.hours + b2 * data.educ)

    # Outcome as Bernoulli rv with success probability
    # given by sigmoid function conditioned on actual data
    pm.Bernoulli(name='logit', p=likelihood, observed=data.income)
```

Visualization and plate notation

The `pm.model_to_graphviz(manual_logistic_model)` command produces the plate notation displayed in the preceding diagram on the right. It shows the unobserved parameters as light and the observed elements as dark circles. The rectangle indicates the number of repetitions of the observed model element implied by the data included in the model definition.

The Generalized Linear Models module

PyMC3 includes numerous common models so that we can usually leave the manual specification for custom applications. The following code defines the same logistic regression as a member of the **Generalized Linear Models** (**GLM**) family using the formula format inspired by the statistical language R that's ported to Python by the `patsy` library:

```
with pm.Model() as logistic_model:
    pm.glm.GLM.from_formula('income ~ hours + educ',
                            data,
                            family=pm.glm.families.Binomial())
```

MAP inference

We obtain point MAP estimates for the three parameters using the just defined model's `.find_MAP()` method:

```
with logistic_model:
    map_estimate = pm.find_MAP()
print_map(map_estimate)
Intercept   -6.561862
hours        0.040681
educ         0.350390
```

PyMC3 solves the optimization problem of finding the posterior point with the highest density using the quasi-Newton **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** algorithm, but offers several alternatives, which are provided by the sciPy library. The result is virtually identical to the corresponding statsmodels estimate (see the notebook for more information).

Approximate inference – MCMC

We will use a slightly more complicated model to illustrate Markov chain Monte Carlo inference:

```
formula = 'income ~ sex + age+ I(age ** 2) + hours + educ'
```

Patsy's function, `I()`, allows us to use regular Python expressions to create new variables on the fly. Here, we square `age` to capture the non-linear relationship that more experience adds less income later in life.

Note that variables measured on very different scales can slow down the sampling process. Hence, we first apply sklearn's `scale()` function to standardize the `age`, `hours`, and `educ` variables.

Once we have defined our model with the new formula, we are ready to perform inference to approximate the posterior distribution. MCMC sampling algorithms are available through the `pm.sample()` function.

By default, PyMC3 automatically selects the most efficient sampler and initializes the sampling process for efficient convergence. For a continuous model, PyMC3 chooses the NUTS sampler that we discussed in the previous section. It also runs variational inference via ADVI to find good starting parameters for the sampler. One among several alternatives is to use the MAP estimate.

To see what convergence looks like, we first draw only `100` samples after tuning the sampler for `1000` iterations. This will be discarded afterwards. The sampling process can be parallelized for multiple chains using the `cores` argument (except when using GPU):

```
with logistic_model:
    trace = pm.sample(draws=100, tune=1000,
                      init='adapt_diag', # alternative initialization
                      chains=4, cores=2,
                      random_seed=42)
```

The resulting trace contains the sampled values for each random variable. We can continue sampling by providing the trace of a prior run as input (see the notebook for more information).

Credible intervals

We can compute the credible intervals—the Bayesian counterpart of confidence intervals—as percentiles of the trace. The resulting boundaries reflect confidence about the range of the parameter value for a given probability threshold, as opposed to the number of times the parameter will be within this range for a large number of trials. The notebook illustrates computation and visualization.

Approximate inference – variational Bayes

The interface for variational inference is very similar to the MCMC implementation. We just use the `fit()` function instead of the `sample()` function, with the option to include an early stopping `CheckParametersConvergence` callback if the distribution-fitting process converged up to a given tolerance:

```
with logistic_model:
    callback = CheckParametersConvergence(diff='absolute')
    approx = pm.fit(n=100000,
                    callbacks=[callback])
```

We can draw samples from the approximated distribution to obtain a trace object like we did previously for the MCMC sampler:

```
trace_advi = approx.sample(10000)
```

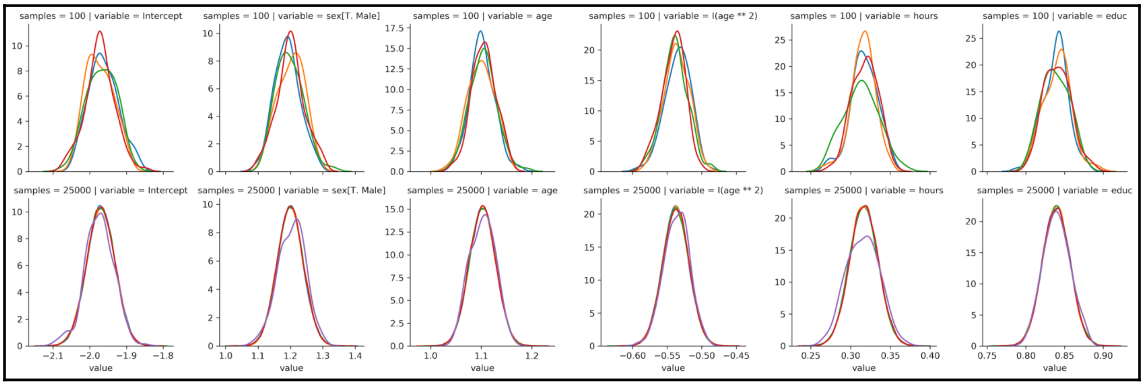
Inspection of the trace summary shows that the results are slightly less accurate.

Model diagnostics

Bayesian model diagnostics includes validating that the sampling process has converged and consistently samples from high probability areas of the posterior, and confirming that the model represents the data well.

Convergence

We can visualize the samples over time and their distributions to check the quality of the results. The following charts show the posterior distributions after an initial 100 and an additional 100,000 samples, respectively, and illustrate how convergence implies that multiple chains identify the same distribution. The `pm.trace_plot()` function shows the evolution of the samples as well (see the notebook for more information):



Posterior distributions

PyMC3 produces various summary statistics for a sampler. These are available as individual functions in the stats module, or by providing a trace to the `pm.summary()` function:

	statsmodels	mean	sd	hpd_2.5	hpd_97.5	n_eff	Rhat
Intercept	-1.97	-1.97	0.04	-2.04	-1.89	69,492.17	1.00

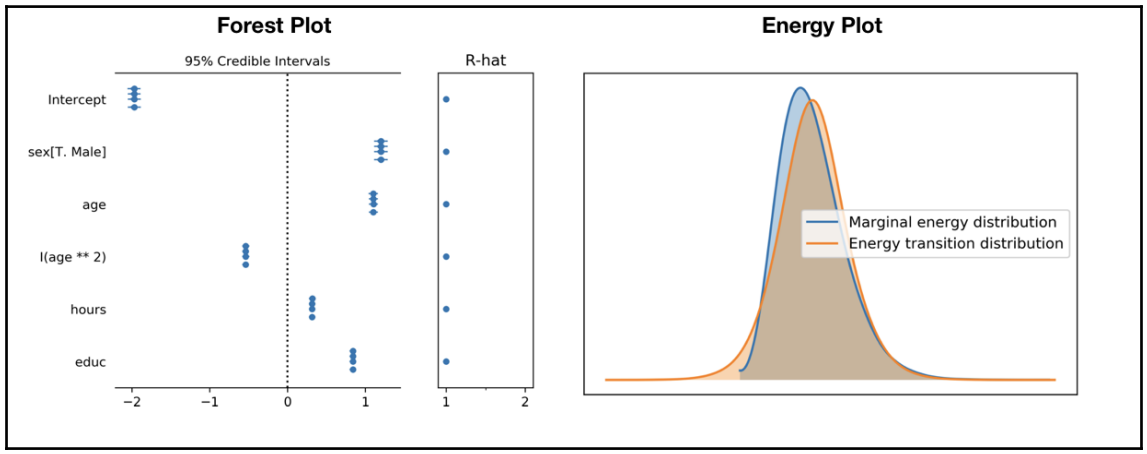
sex[T. Male]	1.20	1.20	0.04	1.12	1.28	72,374.10	1.00
age	1.10	1.10	0.03	1.05	1.15	68,446.73	1.00
I(age ** 2)	-0.54	-0.54	0.02	-0.58	-0.50	66,539.66	1.00
hours	0.32	0.32	0.02	0.28	0.35	93,008.86	1.00
educ	0.84	0.84	0.02	0.80	0.87	98,125.26	1.00

The preceding tables includes the (separately computed) statsmodels `logit` coefficients in the first column to show that, in this simple case, both models agree because the sample mean is very close to the coefficients.

The remaining columns contain the **highest posterior density** (**HPD**) estimate for the minimum width credible interval, the Bayesian version of a confidence interval, which here is computed at the 95% level. The `n_eff` statistic summarizes the number of effective (not rejected) samples resulting from the ~100K draws.

R-hat, also known as the Gelman-Rubin statistic, checks convergence by comparing the variance between chains to the variance within each chain. If the sampler converged, these variances should be identical, that is, the chains should look similar. Hence, the statistic should be near 1. The `pm.forest_plot()` function also summarizes this statistic for the multiple chains (see the notebook for more information).

For high-dimensional models with many variables, it becomes cumbersome to inspect numerous traces. When using NUTS, the energy plot helps to assess problems of convergence. It summarizes how efficiently the random process explores the posterior. The plot shows the energy and the energy transition matrix, which should be well-matched, as in the following example (see references for conceptual detail):



Posterior Predictive Checks

Posterior Predictive Checks (**PPCs**) are very useful for examining how well a model fits the data. They do so by generating data from the model using parameters from draws from the posterior. We use the `pm.sample_ppc` function for this purpose and obtain *n* samples for each observation (the GLM module automatically names the outcome `'y'`):

```
ppc = pm.sample_ppc(trace_NUTS, samples=500, model=logistic_model)
ppc['y'].shape
(500, 29170)
```

We can evaluate the in-sample fit using the auc score, for example, to compare different models:

```
roc_auc_score(y_score=np.mean(ppc['y'], axis=0),
              y_true=data.income)
0.8294958565103577
```

Prediction

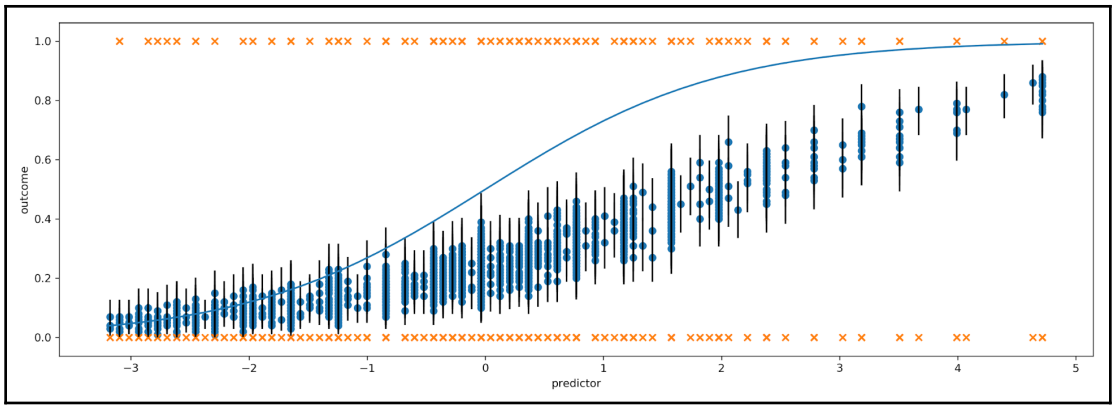
Predictions use Theano's shared variables to replace the training data with test data before running posterior predictive checks. To facilitate visualization, we create a variable with a single predictor hours, create the train and test datasets, and convert the former to a shared variable. Note that we need to use numPy arrays and provide a list of column labels (see the notebook for details):

```
X_shared = theano.shared(X_train.values
with pm.Model() as logistic_model_pred:
    pm.glm.GLM(x=X_shared, labels=labels,
               y=y_train, family=pm.glm.families.Binomial())
```

We then run the sampler as before, and apply the `pm.sample_ppc` function to the resulting trace after replacing the train with test data:

```
X_shared.set_value(X_test)
ppc = pm.sample_ppc(pred_trace, model=logistic_model_pred,
                    samples=100)
```

The AUC score for this model with a single feature is 0.65. The following plot shows the actual outcomes and uncertainty surrounding the predictions for each sampled predictor value:



We will now illustrate how to apply Bayesian analysis to trading-related use cases.

Practical applications

There are numerous applications to Bayesian machine learning methods to investment. The transparency that probabilistic estimates create are naturally useful for risk management and performance evaluation. We will illustrate the computation and comparison of a metric like the Sharpe ratio. The GitHub repository also includes two notebooks referenced below that present the use of Bayesian ML for modeling linear time series and stochastic volatility.

These notebooks have been adapted from tutorials created at Quantopian where Thomas Wiecki leads data science and has significantly contributed to popularizing the use of Bayesian methods. The references also include a tutorial on using Bayesian ML to estimate pairs trading hedging ratios.

Bayesian Sharpe ratio and performance comparison

In this section, we will illustrate how to define the Sharpe ratio as a probability model and compare the resulting posterior distributions for different return series. The Bayesian estimation for two groups provides complete distributions of credible values for the effect size, group means and their difference, standard deviations and their difference, and the normality of the data.

Key use cases include the analysis of differences between alternative strategies, or between a strategy's in-sample return in relation to its out-of-sample return (see the `bayesian_sharpe_ratio` notebook for details). The Bayesian Sharpe ratio is also part of pyfolio's Bayesian tearsheet.

Model definition

To model the Sharpe ratio as a probabilistic model, we need the priors about the distribution of returns and the parameters that govern this distribution. The student t distribution exhibits fat tails that are relative to the normal distribution for low **degrees of freedom** (**df**), and is a reasonable choice to capture this aspect of returns.

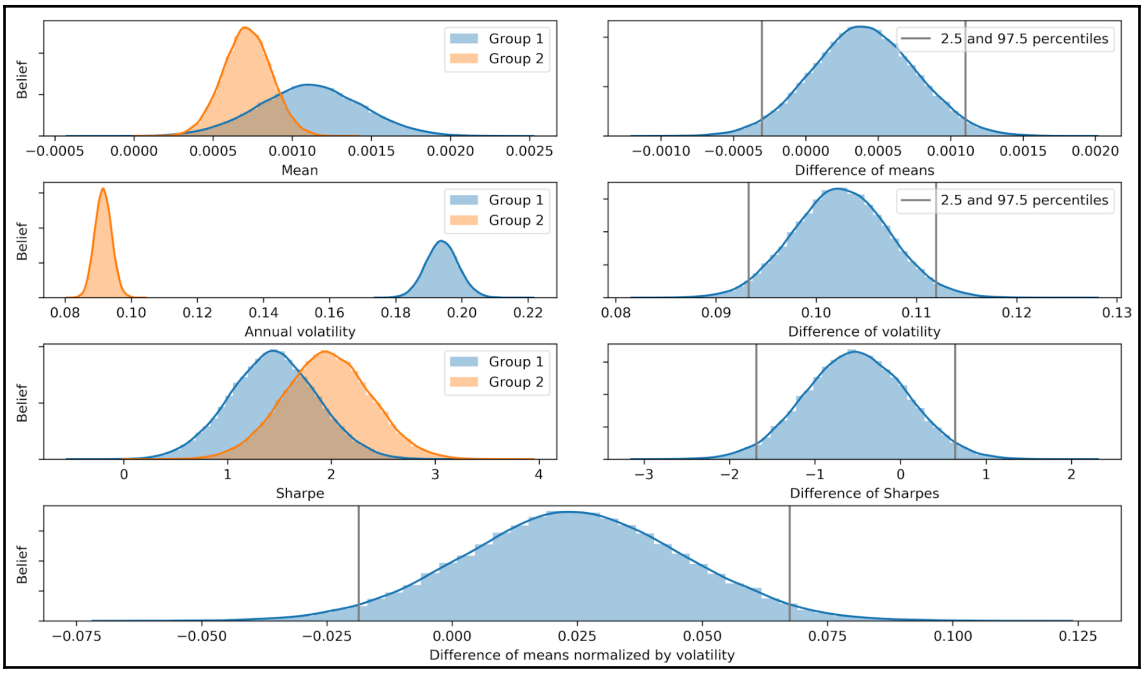
Hence, we need to model the three parameters of this distribution, namely the mean and standard deviation of returns, and the degrees of freedom. We'll assume normal and uniform distributions for the mean and the standard deviation, respectively, and an exponential distribution for the df with a sufficiently low expected value to ensure fat tails. Returns are based on these probabilistic inputs, and the annualized Sharpe ratio results from the standard computation, ignoring a risk-free rate (using daily returns):

```
mean_prior = data.stock.mean()
std_prior = data.stock.std()
std_low = std_prior / 1000
std_high = std_prior * 1000

with pm.Model() as sharpe_model:
    mean = pm.Normal('mean', mu=mean_prior, sd=std_prior)
    std = pm.Uniform('std', lower=std_low, upper=std_high)
    nu = pm.Exponential('nu_minus_two', 1 / 29, testval=4) + 2.
    returns = pm.StudentT('returns', nu=nu, mu=mean, sd=std,
observed=data.stock)

    sharpe = returns.distribution.mean / returns.distribution.variance **
.5 * np.sqrt(252)
    pm.Deterministic('sharpe', sharpe)
```

The notebook contains details on sampling and evaluating the Sharpe ratio for a single stock.

Performance comparison

To compare the performance of two return series, we model each group's Sharpe ratio separately and compute the effect size as the difference between the volatility-adjusted returns. Visualizing the traces reveals granular performance insights into the distributions of each metric, as illustrated by the following chart:



Bayesian Linear Regression for Pairs Trading

In the last chapter, we introduced pairs trading as a popular algorithmic trading strategy that relies on the cointegration of two or more assets. Given such assets, we need to estimate the hedging ratio to decide on the relative magnitude of long and short positions. A basic approach uses linear regression.

The `linear_regression` notebook illustrates how Bayesian linear regression tracks changes in the relationship between two assets over time.

Bayesian time series models

PyMC3 includes AR(p) models that allow us to gain similar insights into the parameter uncertainty, as for the previous models. The `bayesian_time_series` notebook illustrates a time series model for one or more lags.

Stochastic volatility models

As discussed in the last chapter, asset prices have time-varying volatility. In some periods, returns are highly variable, while in others, they are very stable. Stochastic volatility models model this with a latent volatility variable, which is modeled as a stochastic process. The No-U-Turn sampler was introduced using such a model, and the `stochastic_volatility` notebook illustrates this use case.

Summary

In this chapter, we explored Bayesian approaches to machine learning. We saw that they have several advantages, including the ability to encode prior knowledge or opinions, deeper insights into the uncertainty surrounding model estimates and predictions, and the suitability for online learning, where each training sample incrementally impacts the model's prediction.

We learned to apply the Bayesian workflow from model specification to estimation, diagnostics, and prediction using PyMC3 and explored several relevant applications. We will encounter more Bayesian models in `Chapter 14`, *Topic Modeling* and in Chapter 19 on unsupervised deep learning where we will introduce variational autoencoders.

The next two chapter introduce tree-based, non-linear ensemble models, namely random forests and gradient boosting machines.