10
Bayesian ML – Dynamic Sharpe Ratios and Pairs Trading

In this chapter, we will introduce Bayesian approaches to **machine learning** (**ML**) and how their different perspective on uncertainty adds value when developing and evaluating trading strategies.

Bayesian statistics allows us to quantify 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 are fewer 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*, *Portfolio Optimization and Performance Evaluation*) can be interpreted as a Bayesian model. It computes the expected return of an asset 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:

- How Bayesian statistics apply to ML
- Probabilistic programming with PyMC3
- Defining and training ML models using PyMC3
- How to run state-of-the-art sampling methods to conduct approximate inference
- Bayesian ML applications to compute dynamic Sharpe ratios, dynamic pairs trading hedge ratios, and estimate stochastic volatility

You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

How Bayesian machine learning works

Classical statistics is said to follow the frequentist approach 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 percent 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, thus, leaves more room for subjective views and 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 assumes that data is a random sample from a population and aims to identify the fixed parameters that generated the data. Bayesian statistics, in turn, takes 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 well suited for online learning from 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 3 months. In each case, there is both relevant historical data as well as unique circumstances that unfold as the event approaches.

We will first introduce Bayes' theorem, which crystallizes the concept of updating beliefs by combining prior assumptions with new empirical evidence, and compare the resulting parameter estimates with their frequentist counterparts. We will then demonstrate two approaches to Bayesian statistical inference, namely conjugate priors and approximate

inference, which produce insights into the posterior distribution of latent (that is, unobserved) parameters, such as the expected value:

- **Conjugate priors** facilitate the updating process by providing a closed-form solution that allows us to precisely compute the solution. However, such exact, analytical methods are not always available.
- **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

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

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 preceding Keynes quotation captures that spirit. It relies on the conditional and total probability and the chain rule; see Bishop (2006) and Gelman et al. (2013) for an introduction and more.

The probabilistic belief concerns a single parameter or a vector of parameters θ (also: 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 to 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 *Figure 10.1*:

- The **prior** distribution indicates how likely we consider each possible hypothesis.

- The **likelihood function** outputs the probability of observing a dataset when given certain values for the parameters θ, that is, for a specific hypothesis.
- 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.

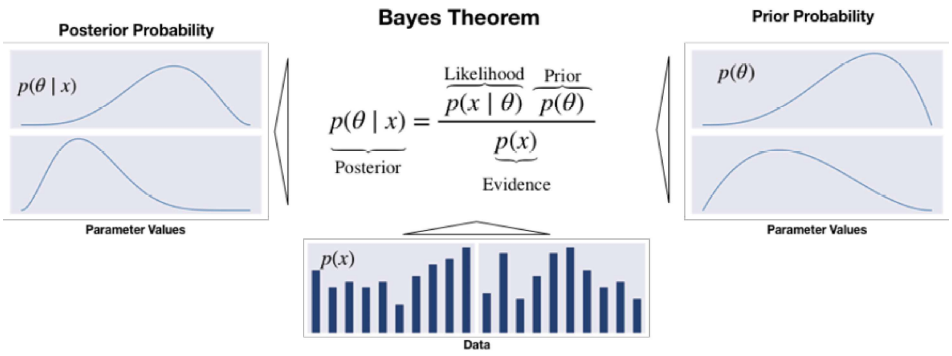

Figure 10.1: How evidence updates the prior to the posterior probability distribution

The posterior is the product of prior and likelihood, divided by the evidence. Thus, it reflects the probability distribution of the hypothesis, updated by taking into account both prior assumptions and the data. Viewed differently, the posterior probability results from applying the chain rule, which, in turn, factorizes the joint distribution of data and parameters.

With higher-dimensional, continuous variables, the formulation becomes more complex and involves (multiple) integrals. Also, an alternative formulation uses odds to express the posterior odds as the product of the prior odds, times the likelihood ratio (see Gelman et al. 2013).

Exact inference – maximum a posteriori estimation

Practical applications of Bayes' rule to exactly compute posterior probabilities are quite limited. This is 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 the fact 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 that 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 will differ from its MLE counterpart:

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

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 to Bayesian models. We will now introduce some convenient choices that facilitate analytical inference.

How to select priors

The prior should reflect knowledge about 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 external to the model into the estimate. In the Black-Litterman context, the investor's belief about an asset's future return would be an example of a subjective prior.
- An **empirical** prior combines Bayesian and frequentist methods and uses historical data to eliminate subjectivity, for example, by estimating various moments to fit a standard distribution. Using some historical average of daily returns rather than a belief about future returns would be an example of a simple empirical prior.

In the context of an ML 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 instance, 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 class or family of distributions as the prior, except for different parameters. For example, when both the prior and the likelihood are normally distributed, then the posterior is also normally distributed.

The conjugacy of 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 the prior for the next update step.

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

Dynamic probability estimates 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 have been plotted in the following image. 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 percent 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:

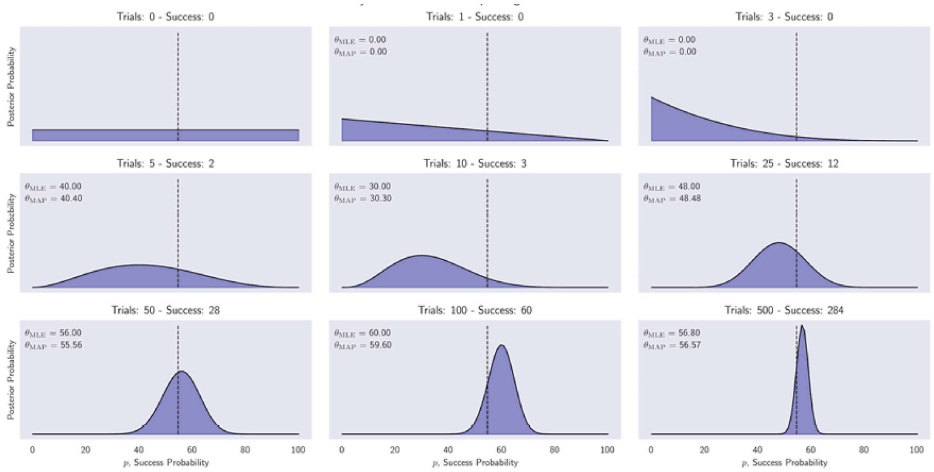

Figure 10.2: Posterior distributions of the probability that the S&P 500 goes up the next day after up to 500 updates

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 a key shortcoming, 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 an approximate rather than exact inference using numerical methods and stochastic simulations, which we will introduce next.

Deterministic and stochastic approximate inference

For most models of practical relevance, it will not be possible to derive the exact posterior distribution analytically and compute 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 that render this calculation prohibitively expensive.

Although for some applications the posterior distribution over unobserved parameters will be of interest, most often, it is primarily required

to evaluate expectations, for example, to make predictions. In such situations, we can rely on approximate inference, which includes stochastic and deterministic approaches:

- **Stochastic** techniques based on **Markov chain Monte Carlo** (**MCMC**) sampling have popularized the use of Bayesian methods across many domains. They generally have the property to converge to the exact result. In practice, sampling methods can be computationally demanding and are often limited to small-scale problems.
- **Deterministic** methods called **variational inference** or **variational Bayes** are based on analytical approximations to the posterior distribution and can scale well to large applications. They make simplifying 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.

We will outline both approaches in the following two sections.

Markov chain MonteCarlo sampling

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 of 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; we will introduce a few popular variations of MCMC-based methods here.

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 that the next step only depends on the current state. In other words, this depends on whether the present, past, and future are 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 do not permit an exact analytic 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 methods aim to identify and explore interesting regions of the posterior that concentrate 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 the process are likely 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 already indicated, 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, which tend to perform better.

Gibbs sampling

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

The algorithm is very simple and easy to implement but produces highly correlated samples that slow down convergence. The sequential nature also prevents parallelization. See Casella and George (1992) for a detailed description and explanation.

Metropolis-Hastings sampling

The Metropolis-Hastings algorithm randomly proposes new locations based on its current state. It does so 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 relative to the corresponding value for the current sample.

A key benefit of the proposal evaluation method is that it works with a proportional rather than an exact evaluation of the posterior. However, it can take a long time to converge. This is 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. See Chib and Greenberg (1995) for a detailed, introductory exposition of the algorithm.

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. With this, it proposes new states for exploration and overcomes some of the MCMC challenges. 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. See Betancourt (2018) for a comprehensive conceptual introduction.

The **No U-Turn Sampler** (**NUTS**, Hoffman and Gelman 2011) 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 and automatic differentiation

Variational inference (**VI**) is an ML 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 better to large data. 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 that is not guaranteed to coincide with the target.

Variational inference is better suited for large datasets, for example, hundreds of millions of text documents, so we can quickly explore many models. In contrast, MCMC will deliver more accurate results on smaller datasets or when time and computational resources pose fewer constraints. For example, MCMC would be a good choice if you had spent 20 years collecting a small but expensive dataset, are confident that your model is appropriate, and you require precise inferences. See Salimans, Kingma, and Welling (2015) for a more detailed comparison.

The downside of variational inference is the need for model-specific derivations and the implementation of a tailored optimization routine, which slows 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 the 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 after Uber open sourced Pyro (based on PyTorch). Google, more recently, added a probability module to TensorFlow.

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

In this section, we will introduce the popular **PyMC3** library, which implements advanced MCMC sampling and variational inference for ML 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), PyMC3 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 used in PyMC2 (released 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 developed at Yoshua Bengio's **Montreal Institute for Learning Algorithms** (**MILA**), which inspired TensorFlow. MILA recently ceased to further develop Theano due to the success of newer deep learning libraries (see *Chapter 16*, *Word Embeddings for Earnings Calls and SEC Filings*, for details).

> PyMC4, released in alpha in December 2019, uses TensorFlow instead of Theano and aims to limit the impact on the API (see the link to the repository on GitHub).

The PyMC3 workflow – predicting a recession

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

1. Encode a probability model by defining:
    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
3. Check your model using various diagnostic tools
4. 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 a simple logistic regression to model the prediction of a recession (see the notebook `pymc3_workflow`). Subsequently, we will use PyMC3 to compute and compare Bayesian Sharpe ratios, estimate dynamic pairs trading ratios, and implement Bayesian linear time-series models.

The data – leading recession indicators

We will use a small and simple dataset so we can focus on the workflow. We will use the **Federal Reserve's Economic Data** (**FRED**) service (see *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*) to download the US recession dates, as defined by the **National Bureau of Economic Research** (**NBER**). We will also source four variables that are commonly used to predict the onset of a recession (Kelley 2019) and available via FRED, namely:

- **The long-term spread of the treasury yield curve**, defined as the difference between the 10-year and the 3-month Treasury yields
- The University of Michigan's **consumer sentiment** indicator
- The **National Financial Conditions Index** (**NFCI**)
- The NFCI **nonfinancial leverage** subindex

The recession dates are identified on a quarterly basis; we will resample all series' frequency to monthly frequency to obtain some 457 observations from 1982-2019. If a quarter is labeled as a recession, we consider all months in that quarter as such.

We will build a model that intends to answer the question: **will the US economy be in recession x months into the future?** In other words, we do not focus on predicting only the first month of a recession; this limits the imbalance to 48 recessionary months.

To this end, we need to pick a lead time; plenty of research has been conducted into a suitable time horizon for various leading indicators: the yield curve tends to send signals up to 24 months ahead of a recession; the NFCI indicators tend to have a shorter lead time (see Kelley, 2019).

The following table largely confirms this experience: it displays the mutual information (see *Chapter 6*, *The Machine Learning Process*) between the binary recession variable and the four leading indicators for horizons from 1-24 months:

Figure 10.3: Mutual information between recession and leading indicators for horizons from 1-24 months

To strike a balance between the shorter horizon for the NFCI indicators and the yield curve, we will pick 12 months as our prediction horizon. The following plots are for the distribution of each indicator, broken down by recession status:

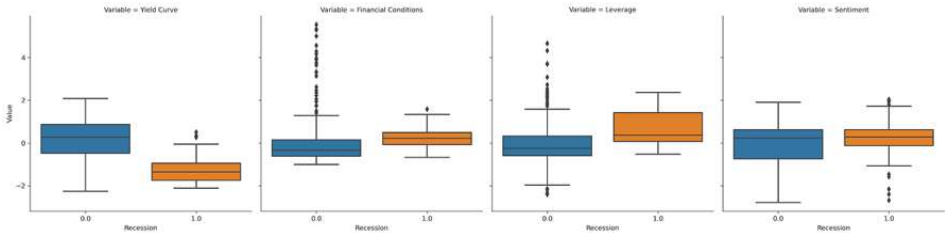

Figure 10.4: Leading indicator distributions by recession status

This shows that recessions tend to be associated with a negative long-term spread of the treasury yield curve, also known as an **inverted yield curve**, when short-term interest rates rise above long-term rates. The NFCI indicators behave as we would expect; the sentiment indicator appears to have the weakest association.

Model definition – Bayesian logistic regression

As discussed in *Chapter 6*, *The Machine Learning Process*, logistic regression estimates a linear relationship between a set of features and a binary outcome, mediated by a sigmoid function to ensure 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 discussed previously, we define the observed RVs via likelihood distributions and unobserved RVs via prior distributions. PyMC3 includes numerous probability distributions for this purpose.

The PyMC3 library makes it very straightforward to perform approximate Bayesian inference for logistic regression. Logistic regression models the probability that the economy will be in recession 12 months after month i based on k features, as outlined on the left side of the following figure:

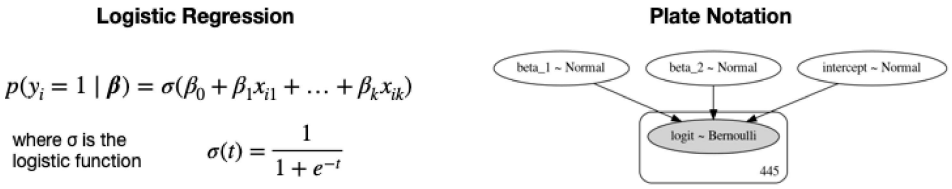

Figure 10.5: Bayesian logistic regression

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 RVs for the unobserved parameters for intercept and two features are expressed using uninformative priors, These 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 the 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)
    beta_1 = pm.Normal('beta_1', 0, sd=100)
    beta_2 = pm.Normal('beta_2', 0, sd=100)
    # Likelihood transforms rvs into probabilities p(y=1)
    # according to logistic regression model.
    likelihood = pm.invlogit(intercept +
                             beta_1 * data.yield_curve +
                             beta_2 * data.leverage)
    # Outcome as Bernoulli rv with success probability
    # given by sigmoid function conditioned on actual data
    pm.Bernoulli(name='logit',
                 p=likelihood,
                 observed=data.recession)
```

Model visualization and plate notation

The command `pm.model_to_graphviz(manual_logistic_model)` produces the plate notation displayed on the right in *Figure 10.5*. It shows the unobserved parameters as light ovals and the observed elements as dark ovals. The rectangle indicates the number of repetitions of the observed model element implied by the data that are included in the model definition.

The generalized linear models module

PyMC3 includes numerous common models so that we can limit the manual specification for custom applications.

The following code defines the same logistic regression as a member of the **Generalized Linear Models** (**GLM**) family. It does so using the formula format inspired by the statistical language R and is ported to Python by the patsy library:

```
with pm.Model() as logistic_model:
    pm.glm.GLM.from_formula(recession ~ yield_curve + leverage,
                            data,
                            family=pm.glm.families.Binomial())
```

Exact MAP inference

We obtain point MAP estimates for the three parameters using the just-defined model's `.find_MAP()` method. As expected, a lower spread value increases the recession probability, as does higher leverage (but to a lesser extent):

```
with logistic_model:
    map_estimate = pm.find_MAP()
print_map(map_estimate)
Intercept     -4.892884
yield_curve   -3.032943
leverage       1.534055
```

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 provided by the SciPy library.

The MAP point estimates are identical to the corresponding `statsmodels` coefficients (see the notebook `pymc3_workflow`).

Approximate inference – MCMC

If we are only interested in point estimates for the model parameters, then for this simple model, the MAP estimate would be sufficient. More complex, custom probabilistic models require sampling techniques to obtain a posterior probability for the parameters.

We will use the model with all its variables to illustrate MCMC inference:

```
formula = 'recession ~ yield_curve + leverage + financial_conditions + sentiment'
with pm.Model() as logistic_model:
    pm.glm.GLM.from_formula(formula=formula,
                            data=data,
                            family=pm.glm.families.Binomial())
# note that pymc3 uses y for the outcome
logistic_model.basic_RVs
[Intercept, yield_curve, leverage, financial_conditions, sentiment, y]
```

Note that variables measured on very different scales can slow down the sampling process. Hence, we first apply the `scale()` function provided by scikit-learn to standardize all features.

Once we have defined our model like this 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 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 1,000 iterations. These will be discarded. 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',
                      chains=4,
                      cores=4,
                      random_seed=42)
```

The resulting `trace` contains the sampled values for each RV. We can inspect the posterior distribution of the chains using the `plot_traces()` function:

```
plot_traces(trace, burnin=0)
```

Figure 10.6 shows both the sample distribution and their values over time for the first two features and the intercept (see the notebook for the full output). At this point, the sampling process has not converged since for each of the features, the four traces yield quite different results; the numbers shown vertically in the left five panels are the averages of the modes of the distributions generated by the four traces:

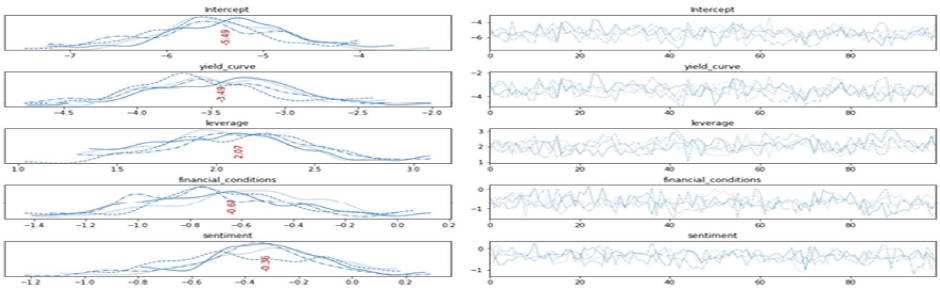

Figure 10.6: Traces after 100 samples

We can continue sampling by providing the trace of a prior run as input. After an additional 20,000 samples, we observe a much different picture, as shown in the following figure. This shows how the sampling process is now much closer to convergence. Also, note that the initial coefficient point estimates were relatively close to the current values:

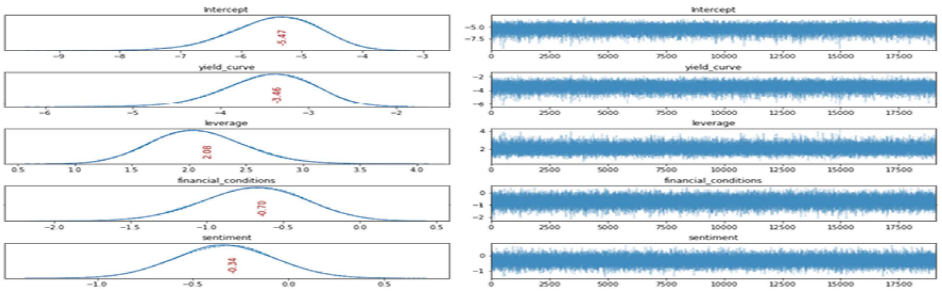

Figure 10.7: Traces after an additional 50,000 samples

We can compute the **credible intervals**, the Bayesian counterpart of confidence intervals, as percentiles of the trace. The resulting boundaries reflect our 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. *Figure 10.8* shows the credible intervals for the variables' yield curve and leverage, expressed in terms of the odds ratio that results from raising *e* to the power of the coefficient value (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*).

See the notebook `pymc3_workflow` for the implementation:

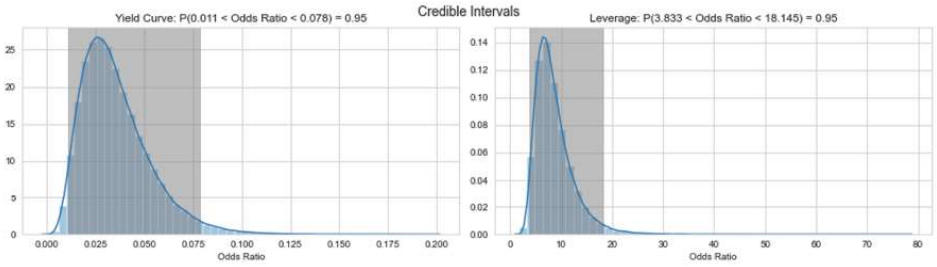

Figure 10.8: Credible intervals for yield curve and leverage

Approximate inference – variational Bayes

The interface for variational inference is very similar to the MCMC implementation. We just use `fit()` instead of the `sample()` function, with the option to include an early stopping `CheckParametersConvergence` callback if the distribution-fitting process converges 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, as 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, as well as 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 charts shown in the following image show the posterior distributions after an initial 100 and an additional 200,000 samples, respectively, and illustrate how convergence implies that multiple chains identify the same distribution:

Figure 10.9: Traces after 400 and after over 200,000 samples

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 function `pm.summary()`.

The following table includes the (separately computed) statsmodels logit coefficients in the first column to show that, in this simple case, both models slightly agree because the sample mean does not match the coefficients. This is likely due to the high degree of quasi-separation: the yield curve's high predictability allows for the perfect prediction of 17 percent of the data points, which, in turn, leads to poorly defined MLE estimates for the logistic regression (see the statsmodels output in the notebook for more information):

Parameters	statsmodels	PyMC3					
	Coefficients	Mean	SD	HPD 3%	HPD 97%	Effective Samples	R hat
Intercept	-5.22	-5.47	0.71	-6.82	-4.17	68,142	1.00
yield_curve	-3.30	-3.47	0.51	-4.44	-2.55	70,479	1.00
leverage	1.98	2.08	0.40	1.34	2.83	72,639	1.00
financial_conditions	-0.65	-0.70	0.33	-1.33	-0.07	91,104	1.00
sentiment	-0.33	-0.34	0.26	-0.82	0.15	106,751	1.00

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 percent level. The `n_eff` statistic summarizes the number of effective (not rejected) samples resulting from the ~100,000 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.

For high-dimensional models with many variables, it becomes cumbersome to inspect numerous traces. When using NUTS, the energy plot helps us 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 example shown in the right-hand panel of the following image:

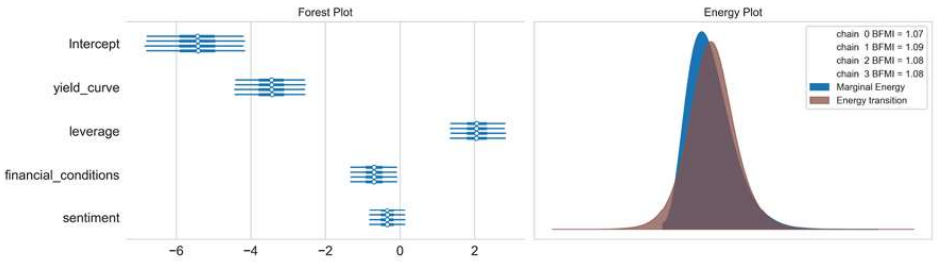

Figure 10.10: Forest and energy plot

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 function `pm.sample_ppc` 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, 445)
```

We can evaluate the in-sample fit using the area under the receiver-operating characteristic curve (AUC, see *Chapter 6*, *The Machine Learning Process*) score to, for example, compare different models:

```
roc_auc_score(y_score=np.mean(ppc['y'], axis=0),
              y_true=data.income)
0.9483627204030226
```

The result is fairly high at almost 0.95.

How to generate predictions

Predictions use Theano's *shared variables* to replace the training data with test data before running posterior predictive checks. To allow for visualization and to simplify the exposition, we use the yield curve variable as the only predictor and ignore the time-series nature of our data.

Instead, we create the train and test sets using scikit-learn's basic `train_test_split()` function, stratified by the outcome, to maintain the class imbalance:

```
X = data[['yield_curve']]
labels = X.columns
y = data.recession
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42,
                                                    stratify=y)
```

We then create a shared variable for that training set, which we replace with the test set in the next step. Note that we need to use NumPy arrays and provide a list of column labels:

```
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 we did previously:

```
with logistic_model_pred:
    pred_trace = pm.sample(draws=10000,
                           tune=1000,
                           chains=2,
                           cores=2,
                           init='adapt_diag')
```

Now, we substitute the test data for the train data on the shared variable and apply the `pm.sample_ppc` function to the resulting `trace`:

```
X_shared.set_value(X_test)
ppc = pm.sample_ppc(pred_trace,
                    model=logistic_model_pred,
                    samples=100)
y_score = np.mean(ppc['y'], axis=0)
roc_auc_score(y_score=np.mean(ppc['y'], axis=0),
              y_true=y_test)
0.8386
```

The AUC score for this simple model is 0.86. Clearly, it is much easier to predict the same recession for another month if the training set already includes examples of this recession from nearby months. Keep in mind that we are using this model for demonstration purposes only.

Figure 10.11 plots the predictions that were sampled from the 100 Monte Carlo chain and the uncertainty surrounding them, as well as the actual binary outcomes and the logistic curve corresponding to the model predictions:

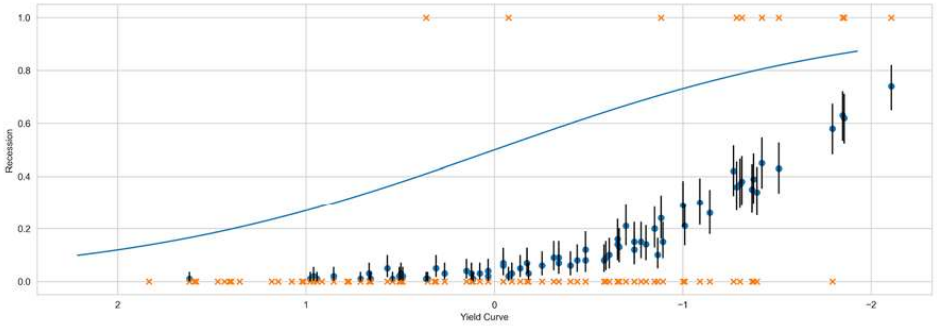

Figure 10.11: Single-variable model predictions

Summary and key takeaways

We have built a simple logistic regression model to predict the probability that the US economy will be in recession in 12 months using four leading indicators. For this simple model, we could get exact MAP estimates of the coefficient values, which we could then use to parameterize the model and make predictions.

However, more complex, custom probability models will not allow for this shortcut, and MAP estimates also do not generate insight into the posterior distribution beyond the point estimate. For this reason, we demonstrated how to run approximate inference using PyMC3. The results illustrated how we learn about the posterior distribution for each of the model parameters, but also showed that even for a small model, the computational cost increases considerably compared to statsmodels MLE estimates. Nonetheless, for sophisticated probabilistic models, sampling-based solutions are the only way to learn about the data.

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

Bayesian ML for trading

Now that we are familiar with the Bayesian approach to ML and probabilistic programming with PyMC3, let's explore a few relevant trading-related applications, namely:

- Modeling the Sharpe ratio as a probabilistic model for more insightful performance comparison
- Computing pairs trading hedge ratios using Bayesian linear regression
- Analyzing linear time series models from a Bayesian perspective

Thomas Wiecki, one of the main PyMC3 authors who also leads Data Science at Quantopian, has created several examples that the following sections follow and build on. The PyMC3 documentation has many additional tutorials (see GitHub for links).

Bayesian Sharpe ratio for performance comparison

In this section, we will illustrate:

- How to define the **Sharpe Ratio** (**SR**) as a probabilistic model using PyMC3
- How to compare its posterior distributions for different return series

The Bayesian estimation for two series offers very rich insights because it provides the complete distributions of the credible values for the effect size, the group SR means and their difference, as well as standard deviations and their difference. The Python implementation is due to Thomas Wiecki and was inspired by the R package BEST (Meredith and Kruschke, 2018).

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

Defining a custom probability model

To model the SR 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 relative to the normal distribution for low **degrees of freedom** (**DF**), and is a reasonable choice to capture this aspect of returns.

We thus need to **model the three parameters of this distribution**, namely the mean and standard deviation of returns, and the DF. 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.

The returns are based on these probabilistic inputs, and the annualized SR results from the standard computation, ignoring a risk-free rate (using

daily returns). We will provide AMZN stock returns from 2010-2018 as input (see the notebook for more on data preparation):

```
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 plate notation, which we introduced in the previous section on the PyMC3 workflow, visualizes the three parameters and their relationships, along with the returns and the number of observations we provided in the following diagram:

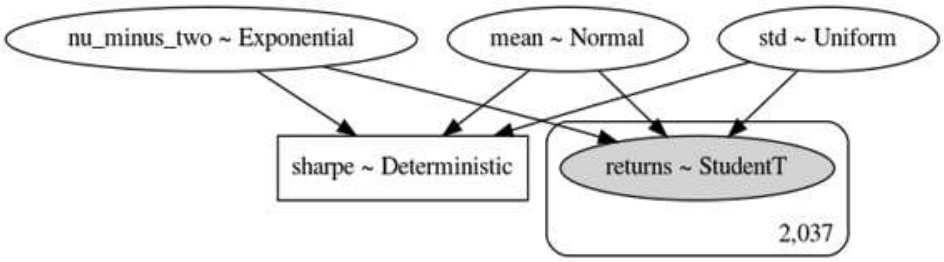

Figure 10.12: The Bayesian SR in plate notation

We then run the MCMC sampling process we introduced in the previous section (see the notebook `bayesian_sharpe_ratio` for the implementation details that follow the familiar workflow). After some 25,000 samples for each of four chains, we obtain the posterior distributions for the model parameters as follows, with the results appearing in the following plots:

```
plot_posterior(data=trace);
```

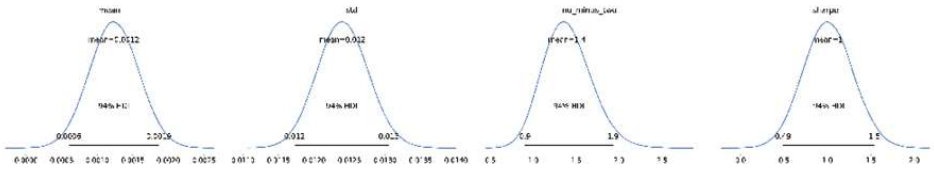

Figure 10.13: The posterior distribution for the model parameters

Now that we know how to evaluate the SR for a single asset or portfolio, let's see how we can compare the performance of two different return series using the Bayesian SR.

Comparing the performance of two return series

To compare the performance of two return series, we will model each group's SR separately and compute the effect size as the difference between the volatility-adjusted returns. The corresponding probability model, displayed in the following diagram, is naturally larger because it includes two SRs, plus their difference:

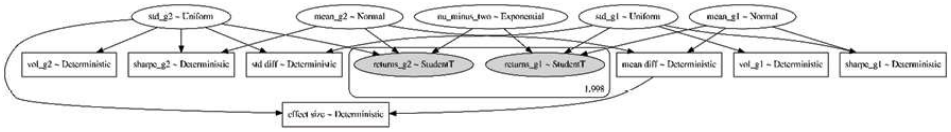

Figure 10.14: The difference between two Bayesian SRs in plate notation

Once we have defined the model, we run it through the MCMC sampling process to obtain the posterior distribution for its parameters. We use 2,037 daily returns for the AMZN stock 2010-2018 and compare it with S&P 500 returns for the same period. We could use the returns on any of our strategy backtests instead of the AMZN returns.

Visualizing the traces reveals granular performance insights into the distributions of each metric, as illustrated by the various plots in *Figure 10.15*:

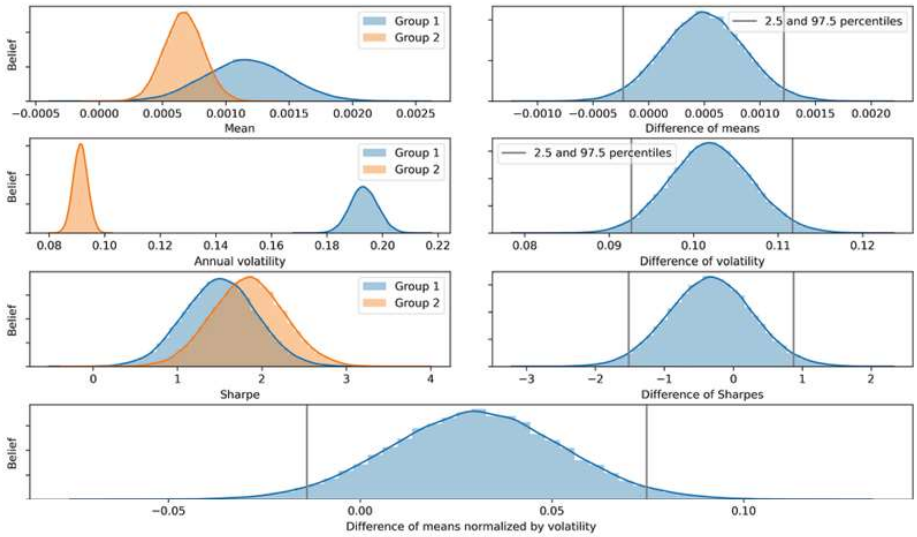

Figure 10.15: The posterior distributions for the differences between two Bayesian SRs

The most important metric is the difference between the two SRs in the bottom panel. Given the full posterior distribution, it is straightforward to visualize or compute the probability that one return series is superior from an SR perspective.

Bayesian rolling regression for pairs trading

In the previous chapter, we introduced pairs trading as a popular 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. You can find the code for this section in the notebook `rolling_regression`, which follows Thomas Wiecki's rolling regression example (see the link to the PyMC3 tutorials on GitHub).

A popular example of pairs trading candidates is ETF GLD, which reflects the gold price and a gold mining stock like GFI. We source the close price data using yfinance for the 2004-2020 period. The left panel of *Figure 10.16* shows the historical price series, while the right panel shows a scatter plot of historical prices, where the hue indicates the time dimension to highlight how the correlation appears to have been evolving. **Note that we should be using the returns**, as we did in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*, to compute the hedge ratio; however, using the prices series creates more striking visualizations. The modeling process itself remains unaffected:

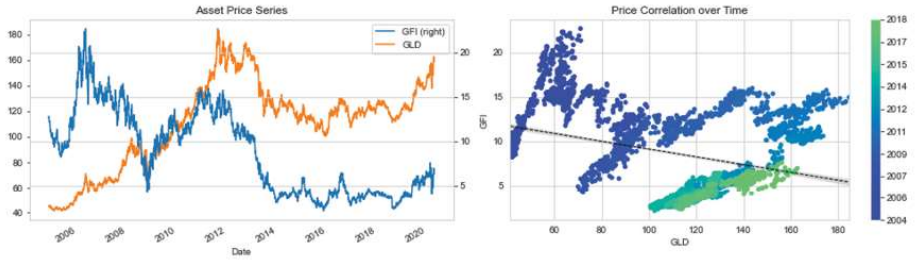

Figure 10.16: Price series and correlation over time of two pairs of trading candidates

We want to illustrate how a rolling Bayesian linear regression can track changes in the relationship between the prices of the two assets over time. The main idea is to incorporate the time dimension into a linear regression by allowing for changes in the regression coefficients. Specifically, we will assume that intercept and slope follow a random walk through time:

We specify `model_randomwalk` using PyMC3's built-in `pm.GaussianRandomWalk` process. It requires us to define a standard deviation for both intercept alpha and slope beta:

```
model_randomwalk = pm.Model()
with model_randomwalk:
    sigma_alpha = pm.Exponential('sigma_alpha', 50.)
    alpha = pm.GaussianRandomWalk('alpha',
                                  sd=sigma_alpha,
                                  shape=len(prices))
    sigma_beta = pm.Exponential('sigma_beta', 50.)
    beta = pm.GaussianRandomWalk('beta',
                                 sd=sigma_beta,
                                 shape=len(prices))
```

Given the specification of the probabilistic model, we will now define the regression and connect it to the input data:

```
with model_randomwalk:
    # Define regression
    regression = alpha + beta * prices_normed.GLD
    # Assume prices are normally distributed
    # Get mean from regression.
```

```
    sd = pm.HalfNormal('sd', sd=.1)
    likelihood = pm.Normal('y',
                           mu=regression,
                           sd=sd,
                           observed=prices_normed.GFI)
```

Now, we can run our MCMC sampler to generate the posterior distribution for the model parameters:

```
with model_randomwalk:
    trace_rw = pm.sample(tune=2000,
                         cores=4,
                         draws=200,
                         nuts_kwargs=dict(target_accept=.9))
```

Figure 10.17 depicts how the intercept and slope coefficients have changed over the years, underlining the evolving correlations:

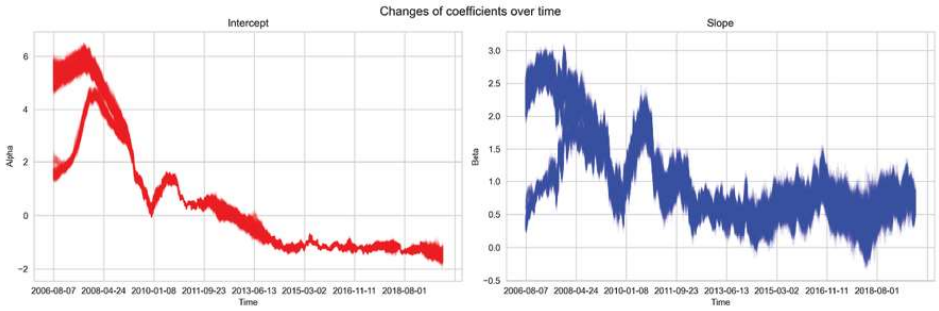

Figure 10.17: Changes in intercept and slope coefficients over time

Using the dynamic regression coefficients, we can now visualize how the hedge ratio suggested by the rolling regression would have changed over the years using this Bayesian approach, which models the coefficients as a random walk.

The following plot combines the prices series and the regression lines, where the hue once again indicates the timeline (view this in the notebook for the color output):

Figure 10.18: Rolling regression lines and price series

For our last example, we'll implement a Bayesian stochastic volatility model.

Stochastic volatility models

As discussed in the previous chapter, asset prices have time-varying volatility. In some periods, returns are highly variable, while in others, they are very stable. We covered ARCH/GARCH models that approach this challenge from a classical linear regression perspective in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*.

Bayesian stochastic volatility models capture this volatility phenomenon with a latent volatility variable, modeled as a stochastic process. The No-U-Turn Sampler was introduced using such a model (Hoffman, et al. 2011), and the notebook `stochastic_volatility` illustrates this use case with daily data for the S&P 500 after 2000. *Figure 10.19* shows several volatility clusters throughout the period:

Figure 10.19: Daily S&P 500 log returns

The probabilistic model specifies that the log returns follow a t-distribution, which has fat tails, as also generally observed for asset returns. The t-distribution is governed by the parameter ν, which represents the DF. It is also called the normality parameter because the t-distribution approaches the normal distribution as ν increases. This parameter is assumed to have an exponential distribution with parameter $\lambda = 0.1$.

Furthermore, the log returns are assumed to have mean zero, while the standard deviation follows a random walk with a standard deviation that also has an exponential distribution:

We implement this model in PyMC3 as follows to mirror its probabilistic specification, using log returns to match the model:

```
prices = pd.read_hdf('../data/assets.h5', key='sp500/prices').loc['2000':,
                                                                  'Close']
log_returns = np.log(prices).diff().dropna()
with pm.Model() as model:
```

```
    step_size = pm.Exponential('sigma', 50.)
    s = GaussianRandomWalk('s', sd=step_size,
                           shape=len(log_returns))
    nu = pm.Exponential('nu', .1)
    r = pm.StudentT('r', nu=nu,
                    lam=pm.math.exp(-2*s),
                    observed=log_returns)
```

Next, we draw 5,000 NUTS samples after a burn-in period of 2,000 samples, using a higher acceptance rate than the default of 0.8, as recommended for problematic posteriors by the PyMC3 docs (see the appropriate links on GitHub):

```
with model:
    trace = pm.sample(tune=2000,
                      draws=5000,
                      nuts_kwargs=dict(target_accept=.9))
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [nu, s, sigma]
Sampling 4 chains, 0 divergences: 100%|██████████| 28000/28000 [27:46<00:00, 16.80dr
The estimated number of effective samples is smaller than 200 for some parameters.
```

After 28,000 total samples for the four chains, the trace plot in the following image confirms that the sampling process has converged:

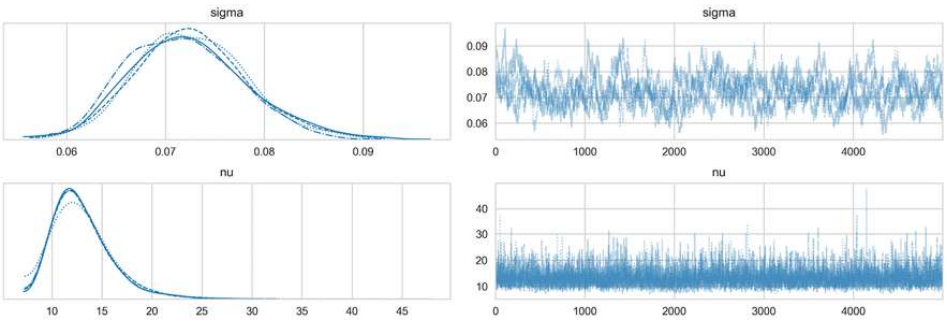

Figure 10.20: Trace plot for the stochastic volatility model

When we plot the samples against the S&P 500 returns in *Figure 10.21*, we see that this simple stochastic volatility model tracks the volatility clusters fairly well:

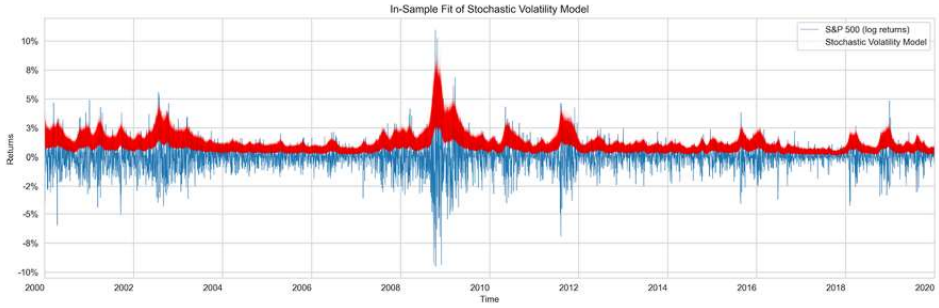

Figure 10.21: Model

Keep in mind that this represents the in-sample fit. As a next step, you should try to evaluate the predictive accuracy. We covered how to make predictions in the previous subsection on rolling linear regression and used time-series cross validation in several previous chapters, which provides you with all the tools you need for this purpose!

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 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*, *Text Data for Trading – Sentiment Analysis*, where we'll discuss natural language processing and topic modeling, and in *Chapter 20*, *Autoencoders for Conditional Risk Factors and Asset Pricing*, where we'll introduce variational autoencoders.

The next chapter introduces nonlinear, tree-based models, namely decision trees, and shows how to combine multiple models into an ensemble of trees to create a random forest.