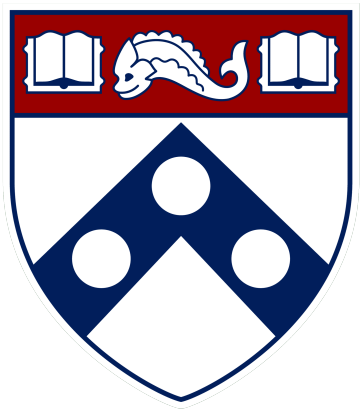
Probabilistic Programming Languages for Performing Bayesian Inference in Sports Futures Prediction

Krish Shah
ksh2002@wharton.upenn.edu



Thesis Advisor: Professor Abraham Wyner
ajw@wharton.upenn.edu
Course Advisor: Professor Norman Badler
badler@seas.upenn.edu

University of Pennsylvania, School of Engineering and Applied Science
CIS 4980 Senior Thesis, Spring 2024

ABSTRACT

We investigate probabilistic programming and its applications in sports analytics, in particular the development of Bayesian models for futures markets prediction. We first review topics in Bayesian statistics, providing theoretical foundation for readers unfamiliar with the field. We establish probabilistic programming as a preferred medium for developing probabilistic models, highlighting their frameworks for reasoning about uncertainty, executing Bayesian inference and generating more interpretable probabilistic models. We then use probabilistic programming to develop a model for NBA futures prediction by repeated sampling from an estimated posterior distribution and compare this model to non-Bayesian accepted practices. While certain hyperparameter settings can make these methodologies roughly equivalent, the Bayesian methodology is more robust to initial hyperparameter settings.

TABLE OF CONTENTS

Motivation

There is an increasing need for sophisticated and interpretable models across various domains where uncertainty and variability are prevalent. Traditional methods often struggle to handle these complexities, leading to predictions that can be inconsistent or difficult to interpret. Bayesian inference offers a powerful alternative by providing a rigorous framework for incorporating prior knowledge and updating beliefs with new data. This approach not only improves the accuracy of predictions but also enhances their interpretability and robustness, making it easier for analysts and stakeholders to understand and trust the results.

Probabilistic programming has emerged as a key tool in expanding the use of Bayesian inference in practical applications. It simplifies the process of developing and implementing complex probabilistic models, allowing researchers to focus on the conceptual aspects rather than the computational intricacies. By leveraging probabilistic programming, we can efficiently perform Bayesian inference, generate posterior distributions, and conduct robust analyses under various scenarios. As a motivating example, we demonstrate the advantages of probabilistic programming in developing a Bayesian model for NBA futures prediction, showcasing its potential to outperform traditional methods.

CHAPTER 1

Bayesian Inference

This section covers and summarises concepts in Bayesian statistics and inference.

1.1 Probability Models

Probability models quantify a distribution of outcomes by assigning a probability to every outcome in the space of possible outcomes using a probability measure P.

This measure requires that

1. $P(\ ) = 1$
2. $P(;) = 0$
3. $8\omega\ 2\ \ , P(\omega)\ 2$ [0..1]
4. For any finite union $A = A_1\ [\ A_2\ [\ \ \ [\ A_n$ of disjoint events on , $P(A) = \sum_{A_i 2 A} P(A_i)$

This contrasts with many traditional statistical techniques that may not explicitly model the uncertainty or the random nature of data but rather focus on summarizing data trends and relationships. For instance, regression analysis predicts the relationship between variables, often focusing on estimation and inference without directly modeling the underlying probability distributions of error terms or residuals. Further, probability models emphasize understanding the processes that generate data. This is achieved by constructing models that explicitly describe the probability of observing data given certain conditions or parameters. On the other hand, non-probabilistic statistical methods focus more on the correlation between variables rather than on describing how data is generated. Because of this, probability models are often more interpretable. Probability models also provide a more direct way to quantify uncertainty by explicitly assigning probabilities to all outcomes, whereas many machine learning techniques provide point estimates or classifications without a direct measure of uncertainty.

Probability models can be both parametric and non-parametric. Parametric models assume that the data adheres to a specific distributional form characterized by a finite set of parameters. This assumption is mathematically convenient, as it often leads to models that are easier to analyze and compute. Non-parametric models don't make these assumptions. This makes these models more flexible – which can allow for easier iteration/augmentation and modelling a wider range of distributions but also increases the di□culty of estimation.

1.2 Bayesian Statistics

Bayesian statistics is motivated from a perspective that uncertainty in unobservable (latent) parameters can and should be modeled using probability distributions. This approach reflects a belief that our knowledge about these parameters is inherently uncertain. Unlike frequentist statistics, which treats parameters as fixed but unknown values that are to be estimated from the data, Bayesian methods incorporate both prior knowledge and observed data to generate posterior estimates for the distribution of parameter values. In particular, Bayesian models may be preferable to frequentist/classical models when data is scarce or practicioners want to encode specific domain-knowledge.

Moreover, Bayesian statistics not only facilitates the estimation of parameters but also enables direct probability statements about parameters and predictions. This is a distinct advantage over traditional methods, which typically focus on point estimates and frequentist confidence intervals that do not have an equivalent probabilistic interpretation.

1.2.1 Prior Distributions

Prior distributions, $p(\theta)$, encode prior beliefs about the distributions of latent parameters. Priors can be either informative or non-informative. Informative priors are when the practitioner chooses to guide the inference process by specifying the hyperparameters in the prior distribution. These are often used when the practitioner has expert knowledge in the subject domain or there is limited data. Non-informative priors are specified such that there is minimal influence exerted by the prior distribution, and are often used when the practitioner is unwilling/unable to make assumptions about the parameter distribution.

Empirical Bayes is a method that attempts to strike a balance between informative and non-informative priors by estimating the prior distribution directly from the data. This approach starts with a prior distribution that is then adjusted based on the data, leading to a data-driven, yet still Bayesian, estimation process.

1.2.2 Posterior Distributions

After observing new data, in Bayesian statistics the prior distribution is updated to reflect the new information gained, resulting in the posterior distribution, namely $p(\theta|x)$. The posterior predictive distribution is the updated distribution of the outcome of interest $p(y|x)$, which is determined using the model's specified $p(y|\theta)$ and the posterior $p(\theta|x)$.

Conjugacy occurs when the sampled posterior distribution comes from the same family as the prior distribution. Conjugate priors can often enable the model to be evaluated analytically, and are additionally convenient because the prior distribution can simply be interpreted as additional data on top of a non-informative prior (Gelman et al., 2021). Non-conjugate priors may be less interpretable and more computationally expensive to estimate but are theoretically no less e□ective than conjugate priors.

1.2.3 Bayes Rule and MAP Estimation

Bayes Rule is the central tool for transitioning from the prior distribution to the posterior distribution based on the observed data, using the following rule

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)}$$

$p(x)$ can be considered a normalizing constant, so in practice this is often applied as

$$p(\theta|x) \quad p(x|\theta)p(\theta)$$

In classical statistics, maximum likelihood estimation (MLE) finds the parameter values that maximize the likelihood of the observed data. That is,

$$\hat{\theta}_{MLE} = \arg\max p(\theta|x)$$

In Bayesian statistics, maximum a posteriori estimation (MAP) finds the parameter values that maximize the likelihood of the posterior distribution. That is,

$$\hat{\theta}_{MAP} = \arg\max p(\theta|x) = \arg\max(p(x|\theta)p(\theta)) \text{ (applying Bayes)}$$

1.3 Markov Chain Monte Carlo Sampling

In some cases, Bayes' Rule can be used to analytically solve for and evaluate the posterior distribution. When this becomes intractable, alternative sampling mechanisms need to be used in order to sample from the posterior distribution. Markov Chain Monte Carlo sampling is an alternative method for evaluating posterior distributions when analytic evaluation is intractable or ine□cient.

1.3.1 Markov Strong Law of Large Numbers

The Markov Strong Law of Large Numbers (MSLL) states that under certain conditions, for a Markov chain visiting states $S_1, S_2, \ldots S_n$ with stationary distribution π through a function of interest $f(S)$,

$$\lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} f(S_i) = E\ [f(S)] \text{ almost surely}$$

MCMC sampling algorithms work by generating a Markov chain whose stationary distribution is equivalent to the target distribution for sampling (in this case the posterior

distribution). The MSLL provides theoretical justification towards the convergence of this process. With this, once the chain has converged, samples from the chain will be equivalent to samples from the target distribution – which is also the posterior distribution.

1.3.2 Convergence of MCMC Sampling

There are number of diagnostic tools to evaluate convergence of the MCMC sampler. One such measure is the $\hat{R}$ statistic, where compares the variance between multiple MCMC chains to the variance within chains. Let W be the within-chain variance and B be the between-chain variance, then $\hat{R} = \sqrt{\frac{(n-1)W+B}{Wn}}$. When the statistic is approximately one, it indicates that the variance between chains is similar to that between chains, evidence that the chains have converged. The $\hat{R}$ statistic is computed individually for each parameter – as it is possible that some parameters may converge faster than others (Gelman et al., 2021).

Further, as MCMC algorithms sample from the converged chain, there can exist high autocorrelation between consecutive samples. Autocorrelation between samples can violate the MSLL and in turn lead to samples that do not truly reflect the posterior distribution. One method to combat this is thinning, where every kth sample from the chain is used (and the rest discarded) as samples far enough apart in the chain will not be autocorrelated with one another (Gelman et al., 2021).

These issues can exacerbate inefficiencies in the sampling procedure, as the effective (or usable) number of samples from the posterior distribution may be significantly smaller than the total number of samples produced.

1.3.3 Metropolis-Hastings

The Metropolis-Hastings algorithm is a common MCMC method (Gelman et al., 2021), that effectively does a random walk through the parameter space.

Algorithm 1: Metropolis-Hastings Algorithm

Input: Initial guess θ_0, target distribution $p(\theta|x)$, proposal distribution $q(\theta'|\theta)$, number of iterations N

Output: Samples from $p(\theta|x)$

Initialize θ θ_0, samples []

for t = 1 *to* N do

  Sample θ' from $q(\theta'|\theta)$

  Calculate acceptance probability α $\min\left(1, \frac{p(\theta'|x)q(\theta|\theta')}{p(\theta|x)q(\theta'|\theta)}\right)$

  Generate u Uniform$(0, 1)$

  if u α then

    θ θ' (Accept the candidate)

  end

  Store θ in samples

end

return samples

The choice of the proposal distribution $q(\theta'|\theta)$ dictates the effectiveness of the MH algorithm, which will efficiently sample from the posterior distribution if (Gelman et al., 2021):

- It is easy to compute $q(\theta'|\theta)$
- It is easy to compute α
- The sampled θ' moves reasonably far at each step.
- The sampled θ' is reasonably likely to be accepted

1.3.4 Other MCMC Samplers

Slice sampling is a MCMC method that does not require a proposal distribution, which can be appealing when practitioners do not wish to tune a Metropolis-Hastings sampler. Slice samplers effectively slice regions of the target distribution for exploration by choosing a value of x, sampling a y value on the interval $[0, f(x)]$ and then sampling points underneath the density curve above the line-segment y (the slice). Slice sampling uses the property that a distribution can be sampled from by sampling points uniformly from the region under the density curve (Neal, 2003). However, in practice slice sampling performs poorly in high-dimensional problems.

Hamiltonian Monte Carlo (HMC) techniques take steps informed by the gradients of the parameters to reduce the random walk behavior often seen in techniques like Metropolis-Hastings. This can make these techniques especially effective in high dimensions. HMC is sensitive to the tuning parameters specified, and so in practice, the most commonly used HMC sampler is the No-U-Turn Sampler (NUTS). NUTS requires minimal tuning as it estimates the number of steps in the HMC sampler dynamically, while also us-

ing automatic stopping to prevent it from ine□ciently doubling back in its exploration (Ho□man and Gelman, 2011).

1.4 Bayesian Model Evaluation

1.4.1 Prediction Evaluation

Naturally, we can evaluate models by predictive accuracy. When predicting point estimates, measures like mean squared error are appropriate to assess the accuracy of the model. However, when predicting distributions, inferences need to be additionally evaluated on the uncertainty inherent in the distribution. For this reason, the log-likelihood (ie $\log p(y|\theta)$) is often used to assess the fit of the probabilistic prediction (Gelman et al., 2021). Note that this log-likelihood is taken over the predictive distribution, not the posterior distribution because ultimately we are concerned with the model's predictive capabilities on the actual data, not the latent parameters.

Additionally, the posterior predictive p-value is defined by the probability that the posterior predictive distribution is more extreme than the observed distribution over some test statistic T. More specifically, repeatedly draw from the posterior distribution of θ, compute the posterior predictive distribution given that θ_i and evaluate some T where T is any query on the distribution and/or θ_i (for example the mean of the predictive distribution) and compare it to the value of T on the observed data. Then the p-value is the proportion of the time T on the drawn data is more extreme than that of the observed data (Gelman et al., 2021).

1.4.2 Information Criterion

Information criterion are designed to compare the predictive accuracy of models of differing complexity. In particular, information criteria can be helpful when we are unwilling/unable to test models on out-of-sample data from the true data generating process. Models that have more complex structure/higher number of parameters may fit data better by chance due to the larger number of degrees of freedom. Information criterion adjust within-sample predictive accuracy by applying a penalty based on the number of parameters being fit, in an e□ort to correct for this e□ect.

One such measure is the Aikake Information Criterion (AIC), defined as

$$2 \log p(y|\hat{\theta}_{MLE}) + 2k$$

.

AIC is simple to estimate, but when using informative priors or hierarchical structures (which tend to overfit less), the penalty term $2k$ can become excessive and thus AIC struggles where there is strong prior information. Moreover, the posterior distribution on θ is summarized by a single point estimate (Gelman et al., 2021).

Another measure is the Deviance Information Criterion (DIC), defined as

$$2 \log p(y|\mathrm{E}[\theta|y]) + 2\mathrm{Var}(\log p(y|\theta))$$

DIC is a more Bayesian alternative to AIC as it considers the posterior expectation of θ. Alternative formulations for the penalty term exist but all are designed to not strictly rely on the number of parameters used (Gelman et al., 2021). DIC can struggle when the posterior distribution is not well-summarized by the mean.

Other criterion, such as Watanabe-Akaike (WAIC) averages over the entire posterior distribution as opposed to using a single point estimate for θ (Gelman et al., 2021). WAIC and cross-validation based techniques can struggles in cases where there is dependency within data points (such as time-series) as it requires that data be able to be appropriately partitioned.

1.4.3 Robustness

Beyond predictive strength, models should also be robust to changes in the prior distribution and observed data. Probability models, in particular, are unlikely to perfectly capture the true distribution – however, it is of concern if misspecification of a model could lead to large di□erences in inferences. Small changes in the choice of prior distribution should not lead to large changes in the resulting posterior distribution – if so, it is likely that the prior may be too informative. Further, if changes in the observed data lead to large changes in the posterior distribution, it may be that the prior is not strong enough. Bayesian sensitivity analysis aims to answer this question by systematically perturbing the prior distribution and/or observed data and analyzing changes in the resulting posterior distributions and inferences upon it (Gelman et al., 2021).

1.5 Bayesian Deep Learning

Bayesian deep learning aims to pair Bayesian principles of modelling uncertainty with classical deep learning techniques, scaling Bayesian models into large data/high-parameter settings. Previously we noted that Bayesian methods are particularly useful in small data settings, so it is worth discussing why Bayesian methods may still be preferable in cases of deep learning. Even with a large number of training observations, if the number of observations is small relatively to the number of features, Bayesian methodologies may help prevent overfitting. Furthermore, in some settings, practitioners may want to understand the inherent uncertainty in predictions even if deep learning models may be necessary to produce strong forecasts.

1.5.1 Variational Inference

In high-dimensional settings, MCMC methods may not be able to e□ciently sample from the posterior distribution. Variational inference combats this by approximating the poste-

rior distribution using a parameterized distribution, $q(\theta j\phi)$ (Xing, 2015). This transforms the sampling problem into an optimization problem, namely finding the parameters ϕ that makes $q(\theta j\phi)$ most closely resemble the true posterior distribution $p(\theta jx)$.

Thus to perform variational inference we need two things:

1. Define the family of distributions q. The choice of distribution q depends on the specific task where variational inference is being performed but often uses some transformation of Gaussian or Bernoulli distribution.
2. Define an optimization objective relating $q(\theta j\phi)$ to $p(\theta jx)$.

The Kullback-Leibler Divergence (KL Divergence) is a common method for measuring the similarity of two probability distributions,

$$KL(q(\theta j\phi)jjp(\theta jx)) = \int q(\theta j\phi) \log \frac{q(\theta j\phi)}{p(\theta jx)} d\theta$$

Thus minimizing the KL divergence with respect to θ would be a reasonable optimization objective for the variational inference problem. However, this may be intractable because it relies on specifically knowing the true $p(\theta jx)$. As an alternative, the objective will be maximizing the Evidence Lower Bound (ELBO),

$$ELBO = E_{q(\ j\ )}[\log(xj\theta)] \quad KL(q(\theta j\phi)jjp(\theta))$$

The first term in the loss function is the expected likelihood of the observed data under the "fake" distribution q. The second term is the KL divergence between this distribution and the prior distribution, which e□ectively serves as a regularization term in the optimization. Note that despite not directly minimizing the KL divergence, maximizing the ELBO will find the set of parameters θ that would minimize KL divergence (Xing, 2015).

1.5.2 Bayesian Neural Networks

Briefly, we recap the classical feed-forward neural network. A neural network consists of interconnected layers of nodes, where each connection represents a synapse and each node applies a transformation to its input. The basic architecture includes an input layer that receives the raw data, one or more hidden layers that process the data through weighted connections and activation functions, and an output layer that produces the final prediction. The network learns by adjusting the weights of these connections to minimize a given error function (often simply the mean squared error) through backpropagation.

Backpropagation calculates the gradient of the loss function with respect to each weight by applying the chain rule, e□ectively propagating the error backward through the network. These gradients are then used by an optimization algorithm, such as stochastic gradient

descent, to adjust the weights in the direction that reduces the loss. This process is repeated over many iterations, or epochs, until the network converges to a set of weights that achieve the lowest possible loss.

Bayesian Neural Networks (BNNs) expand on classical feed-forward neural networks in the same way that Bayesian models expand on frequentist statistical models, by incorporating uncertainty in parameters. A BNN assumes that the weights in the neural networks are uncertain, assigning a prior distribution on their values. In practice, this is often a simple normal prior. Sampling from the posterior distribution of these parameters given observed data quickly becomes computationally infeasible through MCMC methods. Instead, BNNs use variational inference to approximate the posterior distribution, where the optimization is maximising ELBO through gradient descent on the weights of the neural network. Note that to approximate the gradient, Monte Carlo sampling techniques are used, so BNNs do not avoid sampling altogether.

1.5.3 Mixture Density Networks

For the sake of discussion, we present an alternative to the BNN for modelling arbitrary $p(y|x)$, mixture density networks (MDNs) (Bishop, 1994). Considering a feed-forward neural network with the following specifications:

- The input to the neural network remains x
- Some number of hidden layers with activation functions. A common choice is the Rectified Linear Unit (ReLU) activation function, defined as $f(x) = \max(x, 0)$. The ReLU activation function is used because it enables non-linearity in the model while simultaneously having a simple gradient enabling e□cient training.
- The output layers will describe a Gaussian Mixture Model, with three components, μ (the means), (the standard deviations) and (the probability of each Gaussian in the mixture). Note that an exponential activation may be needed to make sure every value in is positive and a softmax activation may be needed to ensure that is a valid probability distribution.

The loss function of the neural network is the negative log loss, which would correspond to MLE estimation. MDNs maybe preferable to BNNs for a few reasons:

- They can more easily create non-normal shapes – BNNs often struggle to generate shapes that are non-normal.
- They provide the probability density function (as opposed to BNNs which require sampling). This also enables more e□cient resolution of queries on the distribution.

In general, MDNs are a powerful tool whose application in Bayesian contexts requires more research. Notably, MDNs do not use MAP estimation (there is no prior on the neural network parameters) – but are better suited for modelling some $p(y|x)$ than assuming a prior over parameters. This observation will come up again when discussing deep

probabilistic programming. Mixture density networks can also alternatively be used to estimate the posterior distribution $p(\theta jx)$ directly, which can then be used in a typical Bayesian fashion (Burton et al., 2021).

CHAPTER 2

Probabilistic Programming

This section serves as an introduction to probabilistic programming, covering key components of and implementations within probabilistic programming languages.

2.1 Background and Key Components

Probabilistic programming is a paradigm for simplifying, abstracting and automating Bayesian inference. For a statistician with domain-expertise in Bayesian statistics but limited programming background, probabilistic programming allows them to more simply implement theoretical models. For a programmer with limited Bayesian background, probabilistic programming makes complicated statistical models accessible.

2.1.1 Stochastic Inference Problem

We introduce the stochastic inference problem (Roy, 2016). As we noted earlier, the primary operation in Bayesian inference is the conditioning of our distribution of parameter values based on observed data. The algorithm below is a very basic attempt to carry out the conditioning operation.

The inputs to our algorithm will be probabilistic programs `guesser()` and `checker()` that define some probability distribution. We will expand on what these might look like later, but simply, `guesser()` provides a sample and `checker(guess)` returns (potentially probabilistically) whether `guess` is valid or not. We ultimately return a sample from the underlying probability distribution.

Algorithm 2: `condition(guesser, checker)`

Input: Probabilistic programs `guesser()` *!* `sample`, `checker(sample)` *!* `bool`
Output: A sample from the distribution of the program

```
accept = False
while not accept do
    guess = guesser()
    accept = checker(guess)
end
return guess
```

Clearly, this algorithm should enable sampling from any distribution so long as one can define the `guesser()` and `checker()` functions for that distribution. Further, let's establish the parallel between this algorithm and Bayesian inference. The distribution of `guesser()` can be considered the prior distribution. The likelihood of `guess` is the same as the Pr[`checker(guess) = True`]. Then, drawing a sample from `condition` is the same as drawing a sample from the posterior distribution. Consider the probability some value θ will be returned from the stochastic inference problem. This is

Pr[guesser() = θ] Pr[checker(θ) = True] Translating this, the first term is $p(\theta)$ and the second term is $p(x|\theta)$. Therefore the result is $p(\theta|x)$, the posterior distribution (ignoring the normalizing constant).

To illustrate this, we can look at the toy example where we are flipping a (potentially-biased) coin. Our goal is to determine p, the true probability of heads when flipping this coin.

> Let $\overline{x} \in [0,1]^n$ be a sequence of n observations from this coin.
>
> Let's say that our prior assumption is that the coin's true probability is drawn from Beta(1, 1), so it is uniform on [0..1]. Then we can specify guesser() to return a sample from that distribution. Note that we could just as easily specify the coin's true probability as Beta(α, β), or *any* arbitrary distribution such that = [0..1] and have guesser() sample from this distribution. We will revisit this implication later.
>
> Then, we will specify checker(guess) as follows, where we will assume that each flip of the coin is Bernoulli with probability p. :
>
> ```
> def checker(guess):
>     return x̄ = Binomial(guess, n)
> ```
>
> In other words, sample a new vector of n observations using guess and return True if that vector is equal to $\overline{x}$. Once again, our data-generating assumption could take *any* reasonable form.

Note that in this case our distributional assumptions on the coin have a tractable functional form, so we can compare the results of the stochastic inference problem with the theoretical results. Let our observed data $\overline{x}$ = [1, 0, 1, 1, 0, 1, 0, 1, 1, 1]. In theory, the posterior distribution of the coin will be Beta(10, 4), and the output of checker() resembles the posterior closely.

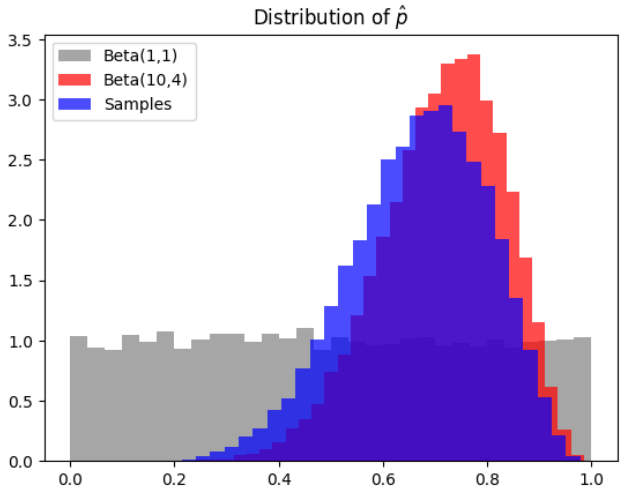


Figure 1: Comparison of `condition()` with Theoretical Posterior

While `condition()` is a good place to start to contextualize the problem, it may not be an e□cient method for sampling. Consider that `condition()` is a geometric process with probability $\Pr[\overline{x}]$ of success, then in expectation it will make $O(\frac{1}{\Pr[\overline{x}]})$ calls to both `guesser()` and `checker()`. In the toy example, randomly choosing `guess` from [0..1] we need roughly 15 million calls to `checker()` to obtain 10,000 valid samples.

When evaluating `checker()` is e□cient and $\Pr[\overline{x}]$ is not too small, `condition` may be e□cient. However, `checker()` can be arbitrarily complex and we don't want to be restricted by $\Pr[\overline{x}]$. Furthermore, in each successive iteration of the loop, the `guesser()` learns nothing new to help it propose a better candidate. We'll discuss later how sampling methods like MCMC are e□cient approximations for `condition()`.

The potential ine□ciency of `condition()` is a large motivation for why probabilistic programming exists (van de Meent et al., 2021). In fact, another explanation of probabilistic programming is to use computer science techniques for algorithm development and implementation to e□ciently approximate `condition()`.

The stochastic inference problem is a form of *backwards* reasoning/computation – given observed data, what can we infer about the underlying parameters of the world? Probabilistic programming languages also support *forwards* reasoning/computation, which works in a generative sense – given our understanding of the underlying parameters, what can we predict about the future? Critically, a single model/object in a probabilistic programming language may need to support both operations.

2.2 Language Architecture

Note that every probabilistic programming language is implemented uniquely. We first touch on some high-level considerations for practitioners deciding which PPL to use, and then touch on common language architecture across PPLs, highlighting interesting and general paradigms used in the language design.

2.2.1 Choosing a Probabilistic Programming Language

There is a wide variety of existing probabilistic programming languages (PPLs); we briefly touch on the high-level distinctions between them. These distinctions impact the performance, usability and applicability of different PPLs. One key distinction lies in their syntax and language design. While some PPLs employ domain-specific languages tailored explicitly for probabilistic modeling, others are embedded within general-purpose programming languages such as Python.

Moreover, PPLs diverge in the range and efficiency of the inference algorithms they offer. Additionally, while some PPLs support both discrete and continuous latent variables, others are tailored for one or the other. This distinction can significantly impact the modeling capabilities of a PPL, particularly when dealing with heterogeneous data types or complex models.

Furthermore, the backend computational frameworks used by PPLs play a crucial role in their scalability and performance. Some languages are built on top of general-purpose numerical libraries like TensorFlow or PyTorch, while others have custom inference engines optimized for specific types of models or inference algorithms. Additionally, factors such as community support, documentation availability, and integration with external libraries and tools can impact the ease of implementation across PPLs.

For our purposes when programming probabilistic programs we will work with PyMC3, a Python-based probabilistic programming language. Working in a language we are already familiar with will allow us to focus on conceptual paradigms over syntax. Further, PyMC3's integration into the existing Python ecosystem makes it flexible and compatible with other data science/statistic techniques. Lastly, PyMC3 offers a wide range of inference algorithms and scales well to large problems.

2.2.2 Language Semantics

In a probabilistic programming languages, distributions are primitives. A distribution can be thought of in terms of its sampling function, (eg. `guesser()`). These kinds of variables can be called stochastic variables, since their values change at runtime. Note that traditional programming languages have stochastic variables too (consider the `rand()` function), but in traditional programming we are often conceptually concerned with its output as opposed to the function itself.

The reason why distributions should be considered primitives is quite intuitive. As we

saw above, the output of the stochastic inference problem is a distribution. In Bayesian statistics, we are interested in distributions (be it a prior distribution, posterior distribution or posterior predictive distribution) and queries on that distribution. If the primary inputs/outputs to our system will be distributions, conceptually we need to consider these primitives in our framework.

There are many frameworks for how a PPL should model these underlying distributional primitives. As a starting point, we discuss a simple version of Sato's distributional semantics (Sato, 1995) that relate logical programming to probabilistic programming.

Consider a finite set F of independent random booleans, probabilistic facts each with their own probability p of being true. Clearly these define a probability distribution over each probabilistic fact (the joint distribution over any truth assignment of the facts can be evaluated as the product of each of the individual truth assignments). Consider further a set of "rules" R that are combinations of facts, other rules and/or their negations. The combination of the set of facts and rules denotes a probabilistic model.

We provide a simple example of such a world here (De Raedt and Kimmig, 2015), denoting the canonical alarm network:

F = {
burgulary, b : 0.1
earthquake, e : 0.2
hearsAlarm(X), $h(X)$: h(mary) = 0.7, h(john) = 0.4
}
R = {
alarm, a : e
alarm, a : b
calls(X), $c(X)$: a ^ $h(X)$
call, c : c(mary) _ c(john)
}

Then consider a "world" where some set of facts and rules are fixed. We can see how forward and backward reasoning would be implemented over this logic (first deterministically over the fixed set and then randomly when considering all possible worlds) (De Raedt and Kimmig, 2015). Consider a set of facts given to us as true {b, h(john)}. Then, b ! a, (a ^ h(john)) ! c(john), c(john) ! c. This deduction represents forward reasoning on this world, and forward reasoning over all worlds (and the likelihood associated with each world) will provide the likelihood of all facts/rules. Given a specific set of observations, we can also deduce the facts that must be true to support this set given

our world. Once again, enumerating all worlds (and their associated likelihood), we can deduce the probability of a given fact/rule, forming backwards reasoning.

This world can be expanded beyond binary choices, though it is worth noting probabilistic facts are sufficient to describe a wide array of Bayesian/Markov-based models (De Raedt and Kimmig, 2015). Consider instead a form of facts where each fact f can take on some value from a set f_S, and each value in f_S occurs with some probability (only one value can be true at once and exactly one must be true at all times). Quickly, it becomes apparent that enumeration over all worlds may be computationally intractable, requiring the use of smarter (and potentially approximate) algorithms.

Beyond this, the probabilistic programming language may specify compositional rules to combine distributions (Dahlqvist et al., 2020). While these may vary from language to language, they will support operations on the joint distribution of individual distributions. In this way, users can define new distributional primitives – either directly through compositional functions or by defining a function that returns samples from the new distribution.

2.2.3 Lazy versus Eager Inference

In traditional programming, the choice between lazy and eager evaluation involves a trade-off between simplicity and efficiency. Eager evaluation computes expressions immediately upon encounter, ensuring that all values are available when needed, simplifying reasoning about program behavior. However, it may lead to unnecessary computations. Lazy evaluation defers computations until results are demanded, potentially conserving resources, but introducing complexity into program behavior. (van de Meent et al., 2021) The preference for eager or lazy evaluation depends on factors such as program nature and desired trade-offs between simplicity and efficiency. Eager evaluation is common in imperative languages for its simplicity, while lazy evaluation, prevalent in functional languages, can enable more efficient resource usage and support advanced programming paradigms.

A similar trade-off exists in probabilistic programming languages between lazy and eager inference (van de Meent et al., 2021). In lazy inference, the system defers the computation of posterior distributions or other inference results until they are explicitly requested by the user or needed for downstream tasks. This can be advantageous in scenarios where not all parts of the model are relevant to the current inference task or when exploring large model spaces where computing all possible inference results upfront would be prohibitively expensive. In fact, for models that have a non-finite grammars (read: “distributional semantic”), lazy evaluation may be required.

While PPLs are designed to carry out inference tasks, much of the code they execute is in fact deterministic. These portions of probabilistic programs have semantics similar to other “traditional" programming languages (Tolpin et al., 2016). Some probabilistic programming languages, including Anglican (a probabilistic programming language integrated through traditional programming language Clojure), use this blend to build smarter

evaluation models. Through its continuation-passing style based evaluation model, Anglican is able to treat deterministic parts of the code conventionally while providing a separate mechanism to handle its probabilistic constructs such as `sample` (Anglican's forward reasoning/sampling) and `observe` (backwards reasoning/conditioning) differently. Specifically, Anglican views calls to these functions as "checkpoints" where normal computation stops and inference algorithms intervene, directing the computation flow based on probabilistic logic. This facilitates efficient execution of probabilistic queries while maintaining the integrity of deterministic computations. This method ensures efficient execution of both deterministic and probabilistic code, preserving the natural syntax and semantics of the underlying traditional programming language while extending it with probabilistic programming capabilities.

2.2.4 Computational Graphs

A trace is record of a probabilistic execution of a probabilistic program (Roy, 2016). It captures the sequence of operations, including the output of random primitives and the resulting control flow decisions from these responses. Traces are often implemented in tree or graph-like data structures, because changing the output of a random primitive higher up in the tree might change the downstream decisions made by a probabilistic program's control flow (Yang et al., 2013). Efficient trace implementation will use evaluation optimizations (like lazy evaluation we discussed earlier) and memoization/compacting representations for samples repeatedly drawn from the same distributional primitive. In theory, the trace is all that is required to run the probabilistic program – if a language allows for the trace to be first recorded, and then replayed, it can execute probabilistic programs. By maintaining traces, probabilistic programs can build more efficient inference algorithms.

One such representation may be through a graph G defined by (V, A, P, Y) (van de Meent et al., 2021):

- V is the set of vertices that represent random variables.
- A is the set of directed edges (v_1, v_2) between random variables that represents conditional dependence from v_1 to v_2.
- P is a function from the set of vertices to the probability mass/density function for that random variable.
- Y is a map from random variables with observations to a (deterministic) expression for those observations.

Because a probabilistic program is ultimately defining a probability distribution, some probabilistic programs can be compiled to the static graph G prior to execution (van de Meent et al., 2021). However, in other cases the graph may not be known at compile time, such as cases where there are random variables dynamically generated during the program's execution.

There are a few methodologies for creating dynamic computation graphs. One method would be to use an upper-bound on the number of objects that can be tracked by the program simultaneously. In this case, the program must also specify which variables need to be tracked at all times (through a separate variable), and the switching of that variable "on/o" will allow a static G to approximate the functionality of a dynamic one (van de Meent et al., 2021). Alternatively, the graph G can be built dynamically during run-time. As the program executes, if additional random variables need to be added to the context then the program will add those nodes to the graph. These methods are known as evaluation-based methods, and do not require G to be defined explicitly at compile-time (or even at all) (van de Meent et al., 2021). These techniques may rely on lazy evaluation since the scope of a random variable's support may not be known until it needs to be evaluated.

2.3 Implementation

Consider the Beta-Binomial model described above and its implementation using traditional Python and PyMC3.

```
# Python
alpha_prior, beta_prior = 1, 1
alpha_post = alpha_prior + np.sum(n_successes)
beta_post = beta_prior + np.sum(n_trials) - np.sum(n_successes)

# PyMC3
with pm.Model() as model:
    p = pm.Beta('p', alpha=alpha_prior, beta=_prior)
    y = pm.Binomial('y', n=n_trials, p=p, observed=n_successes)
    trace = pm.sample(n_samples, return_inferencedata=False)
```

First we can discuss the `sample()` function in PyMC3. This function abstracts the inference algorithms discussed earlier to provide samples from the posterior distribution of the model's parameters given the observed data. Note that `sample()` doesn't require explicit references to the probability model itself, as the code is instantiated within the context of `model`.

At first glance it is unclear why the PyMC3 implementation seems more complicated, and doesn't take advantage of the model's conjugacy. In fact, in this specific setting, the Python code will outperform the PyMC3 implementation. When the exact form of the model is known, the practitioner can specifically tailor their implementation.

However consider minor modifications to this scenario that illustrate the power of the probabilistic programming approach.

1. The practitioner is unaware of or unable to resolve the model analytically, despite an analytical solution existing.

In the above, we assumed that the practitioner has sufficient knowledge of Bayesian statistics to find the analytical form of the posterior distribution for their chosen prior and mixing distribution. This assumption is fairly realistic for a simple case of a Beta-Binomial, but even then creates a barrier to entry for programmers without a strong statistics foundation. Moreover, even for practitioners with strong statistics backgrounds, estimating analytical solutions can be time-consuming. Moreover, some PPLs (although not PyMC3) can recognize conjugacy in certain probabilistic models, and directly apply the analytical solution in those cases.

2. The practitioner wishes to slightly modify either the prior or mixing distribution. For example, they would like the to use a Normal prior instead of the Beta prior. In PyMC3, this is a simple change:

```
# PyMC3
with pm.Model() as model:
    BoundedNormal = pm.Bound(pm.Normal, lower=0, upper=1)
    p = BoundedNormal('p', mu=mean_prior, sd=std_prior)
    y = pm.Binomial('y', n=n_trials, p=p, observed=n_successes)
    trace = pm.sample(n_samples, return_inferencedata=False)
```

Because `sample()` abstracts the inference algorithm, the necessary code change is extremely simple. While no analytical solution may exist for this particular model, the probabilistic programming language returns samples from the posterior distribution using one of many inference algorithms (that could be specified by the user).

Furthermore, `Bound()` is the PyMC3 method to establish a distribution needs to be bounded, which allows it to be a suitable prior on `p`, the parameter for the binomial distribution. The simplicity of this transformation should not understate its usefulness. There is no abstraction for such a distribution in traditional Python, and to use a normal prior one would need to specify some transformation function to map its outputs to [0..1]. The wide range of distributional primitives offered in PyMC3 (and other equivalent probabilistic programming languages) creates numerous potential modifications to even this simple model.

Even within a simple example, it quickly becomes apparent how implementation in a probabilistic programming language can save practitioners time, enable experimentation with non-standard models and simplify the process of structuring probabilistic models.

We can see that even as probabilistic models scale in complexity, probabilistic programming languages allow for relatively simple implementation. Consider the example of a normal-normal hierarchical model. That is, consider each realisation y_{ij} `Normal`(μ_j, σ^2) where μ_j `Normal`(μ, τ^2). Then there are priors on μ `Normal` and τ `HalfNormal` (to enforce it is positive).

```
#PyMC3
with pm.Model() as hierarchical_model:
    mu = pm.Normal('mu', mu=0, sd=10)
    tau = pm.HalfNormal('tau', sd=5)

    mu_j = pm.Normal('mu_j', mu=mu, sd=tau, shape=J)
    y_ij = pm.Normal('y_ij', mu=mu_j, sd=sigma, observed=data)

    trace = pm.sample(1000, return_inferencedata=False, target_accept=0.95)
```

Defining such a model without using probabilistic programming would require defining the log-likelihoods for each of the prior distributions $p(\theta)$, the function $p(xj\theta)$ and the log-likelihood of the posterior distribution $p(\theta jx)$. This would require explicit mathematical definitions for each distributional primitive, and would make changing these primitives di□cult (since it would require permeating the change in the likelihood function across everywhere it was previously defined).

That additional complexity exists before discussing the MCMC sampling algorithm required to sample from the posterior distribution. Without using a probabilistic programming language, the practitioner has to explicitly create a sampling algorithm. This requires not only an understanding of the background statistics of MCMC and MCMC algorithms, but also su□cient programming expertise to e□ciently implement this solution. When naively implemented, sampling algorithms can become ine□cient enough as to become impractical – writing comparably e□cient sampling algorithms in a native programming language requires an understanding of the language's underlying computation and ability for vectorized/parallel computation. Moreover, practitioners lose the flexibility to switch between sampling algorithms depending on use cases. Modern probabilistic programming languages often abstract variational inference as well. In PyMC3, functions like `pm.fit()` abstract the optimization logic required to generate the distribution $q(\theta j\phi)$.

Lastly, evaluating models is much simpler when using a probabilistic programming language. Earlier, we discussed the di□culties in evaluating the output of Bayesian models, both in evaluating the converge of MCMC sampling as well as the model's overall fit. Robust convergence checks are not di□cult to implement (checking for auto-correlation, $\hat{R}$ values and counting e□ective number of samples are simple computational tasks) but can create additional overhead in the modelling process when they exist external to the sampling process. Probabilistic programming languages abstract and automate the evaluation process of Bayesian models within the `sample()` call so that practitioners don't need to verify convergence manually and can be easily alerted to potential sampling issues before moving forward in the modelling process. Furthermore, probabilistic programming languages implement modules for evaluating model performance and comparing models. Specifically in PyMC3, models can compute their element-log likelihood through `pm.compute_log_likelihood()` and can be compared to one another through

`az.compare()`, which computes (among other relevant statistics) the leave out one cross validation score and WAIC. PyMC3 also supports integrations with Python plotting libraries to visualize models and predictions. Once again these abstractions are not complicated but reduce barriers for practitioners to effectively iterate on models.

Probabilistic programming doesn't necessarily enable practitioners to do anything they couldn't do without it – ultimately it is an abstraction and we can see that any model built using a probabilistic programming language could be similarly built without one. However, throughout the life cycle of the model-building process, probabilistic programming reduces barriers to entry, ensures more efficient implementation and enables effective model iteration. In short, probabilistic programming makes using Bayesian models more practical, which will increase their overall use.

2.4 Deep Probabilistic Programming

Thus far, the probabilistic programming paradigms we've discussed are appropriate under a few assumptions (van de Meent et al., 2021):

1. We are capable of specifying a probabilistic model that can accurately model our data.
2. We perform the conditioning operation on a single set of observations at a time, and are concerned with producing the best possible inference on this set of observations.

These assumptions are fairly broad and cover most use cases for traditional Bayesian analysis, which thrives on small datasets (where the use of a prior is necessary to combat variance) or in simulation-based studies where estimating parameter uncertainty is important to capture the correct amount of variance in outcomes (more on this later).

However, in modern machine learning and artificial intelligence tasks, these assumptions can break down (van de Meent et al., 2021). Firstly, there are many datasets for which specifying a realistic probabilistic model may not be possible. Consider tasks like natural language processing or computer vision which operate on datasets like images or language – specifying a probabilistic model over the distribution of images/language may not be feasible. Furthermore, these datasets may not be consistently structured, with data sometimes entirely or partially unlabelled. Secondly, there is a question of scalability. While probabilistic programming languages attempt to efficiently implement inference algorithms, as datasets increase in size and models increase in complexity these algorithms may not scale well.

Deep generative models are similar to the traditional generative models we've discussed previously – they will still perform both forwards and backwards inference. The critical addition is that deep generative models will use neural networks as primitives, in the same way our traditional generative models use generic distributions (van de Meent et al., 2021). These operations are supported by tensors, and most primitive functions in mod-

ern probabilistic programming languages support vectorization that allows them to accept tensors as inputs and outputs. The use of neural networks presents a new problem – parameter estimation. Deep probabilistic programs may have millions of parameters, which is orders of magnitude larger than the models we've discussed. Furthermore, practitioners may lose the advantage of domain-knowledge, previously the structure of our models and parameter estimation could be informed by knowledge of the problem, whereas here parameters must be estimated entirely from data.

The problem of parameter estimation can be solved through inference using Bayesian deep learning. However, this is computationally complex, and implementations may prefer a simpler methodology: stochastic gradient descent. That is, consider some observed data $\overline{Y}$ generated from $\overline{X}$ and parameters θ, and assume that the joint distribution $p(\overline{Y}, \overline{X}, \theta)$ is the product of the individual distributions $p(Y_i, X_i|\theta)$. Then one can use stochastic gradient descent to find the θ that maximises the likelihood of $p(\overline{Y}|\theta)$. Note that this seems like a departure from traditional Bayesian inference, since we no longer specify a prior $p(\theta)$. Specifying a prior may be impractical (because it is an open question of *how* to specify the prior), and in the case where we have a lot of data, the prior term may be dominated anyways (van de Meent et al., 2021).

In stochastic gradient descent, an estimate of the gradient is required to determine the direction to move parameters at each step. Consider $r \log p(Y) = \mathbb{E}_{p(X|Y)}[r \log p(Y, X)]$ (van de Meent et al., 2021). In other words, the gradient of the marginal log likelihood can be computed as an expectation over the posterior distribution. Then, a simple algorithm for performing deep probabilistic programming might perform the typical probabilistic programming inference task of producing samples from the posterior distribution for X, use these samples to estimate the gradient of θ and update θ accordingly via a stochastic gradient descent step (van de Meent et al., 2021). Recall that this framework is essentially performing variational inference – it has changed the sampling task into an optimization task.

Deep probabilistic programming lies at the intersection of traditional machine learning models and Bayesian inference. Further research into e□ciently estimating Bayesian models over large datasets will enable a host of new techniques for machine learning practitioners.

CHAPTER 3

Futures Modelling

This section utilises probabilistic programming language PyMC3 and Bayesian inference techniques to propose a futures modelling technique.

3.1 Problem Description

3.1.1 Overview

A futures market is a market where participants can buy and sell binary contracts that are worth $1 if the event specified in the contract occurs and is worth $0 otherwise. Team futures markets are related to the future performance of a team, including division/conference/championship winner, playo□ likelihood and win totals. While this discussion can apply generally to any futures market, we will focus specifically on NBA win totals.

Define a set of n teams T and a list of games G between two teams (one denoted the home team and the other denoted the away team). Note that G is an ordered list because games are played on a schedule. For every game, there is exactly one winner. We'll let $G_{i;j;x}$ denote the xth time team i plays team j where team i is the home team. For simplicity, assume that every team will play the same number of games, and further that every team will play the same number of home games as away games. Consider some point where $m < jGj$ games have been played in the season. Our goal is to output the joint probability distribution $p(W)$, where W is the vector of total wins for each team after all games in G have been played.

While we only require $p(W)$, we'll look for the joint probability distribution $p(R)$, where R is the result of every game in G. Note that any R determines a W, so $p(R)$ is su□cient for the original task. However, we may prefer to estimate $p(R)$ because it would allow for consistent pricing of other futures markets (such as probability to make playo□s, which would require a knowledge of R).

3.1.2 Key Statistical Properties of Solution

Before developing a methodology for a solution, we will describe some of the statistical characteristics that are necessary for a strong solution.

Firstly, given that we want $p(R)$, our solution must output a probability distribution. This may seem obvious but it is certainly non-trivial. The requirement of a probability distribution eliminates a large swath of potential statistical techniques, and lends itself well towards a probability model. Moreover, estimating an entire probability distribution can quickly become computationally expensive – we require a methodology that can be estimated backwards-looking and predict forwards-looking quickly.

Further, to predict the future performance of a team, we'd like to predict their performance

in future games. Note that while this is not a necessary condition (we could estimate $p(R)$ directly and then use that to estimate the marginal distribution) – it is a simpler problem to predict the outcome of a game given the teams than directly compute the joint distribution. If we can predict the outcome of every game $G_{i;j;x}$ then we can use Monte-Carlo sampling to draw samples from $p(R)$.

Consider then the task of predicting the outcome of an individual game $G_{i;j;x}$. Further, let's assume that the covariate space is t_i, t_j, x_g – a parameter for teams i, j and another parameter for any extraneous game variables. Then we have a function $f : (T, T, G) \; ! \; [0..1]$ that maps the covariate space to a probability of team i winning $G_{i;j;x}$. If we could know t_i, t_j (ie., we know exactly how good every team is) then the accuracy of the predictor would come down to the accuracy of the function f. However, t_i and t_j need to be estimated – specifically from data from previous games. Yet recall that we assume that each individual game result is the outcome of some random process. Therefore our estimates for t_i, t_j are conditional on the outcome of this random process. This introduces uncertainty into the latent parameter estimates. Therefore for $G_{i;j;x}$ we need an understanding of $p(t_i|G), p(t_j|G)$, which is equivalent the posterior distribution for t_i and t_j given the observed data.

3.2 Literature Review

We examine a few methods for estimating the strength parameters t_i, t_j.

3.2.1 Bradley-Terry Models

Generically, the Bradley-Terry methodology assumes that

$$\Pr[G_{i;j;x} = i | t_i, t_j, x_g] = \frac{t_i}{t_i + t_j}, s.t. \sum_i t_i = 1, t_i, t_j > 0$$

. (Hamilton et al., 2023)

This is commonly reformulated as

$$\Pr[G_{i;j;x} = i | t_i, t_j, x_g] = \frac{1}{1 + e^{\left(\frac{(t_i \; t_j)}{B} + S + x_g^T\right)}}$$

. (Zanco et al., 2024)

There are a few things to note here. The Bradley-Terry model assumes a logistic win-probability relationship. The use of the logistic link contrasts to alternative approaches such as Thurstone Models that assume a normal relationship (Thurstone, 1927). This is a nice relationship for a few reasons. Firstly, the logistic function is bounded between 0 and 1, so its output is already on a probability scale. Secondly, the logistic function's shape matches our natural intuition that differences in team-strength should matter closer to 50-

50 than in the tails. That is, win-probability is not a linear relationship on the di□erence in strength, an incremental improvement in team quality should have a larger impact when teams are evenly matched than when they are already mismatched. S and B are arbitrary tuning parameters to scale the team strength parameters. Notably, B allows one to change the base of the logistic-link. Lastly, $x_g^T \lambda$ allows for the inclusion of game-state covariates, including home-court advantage.

Given this functional form, the parameters t_i, t_j can be estimated by maximum likelihood estimation across the set of observed games. Notably, there is no requirement that we have priors on t_i, t_j, but if there were, we may be able to estimate their posterior distributions, using MAP estimation.

3.2.2 Elo Models

The Elo rating system is an alternative methodology for computing relative strength parameters in two-player games. The Elo system assumes the same implied win-probability relationship as the Bradley-Terry model (Zanco et al., 2024), however its method of estimation is di□erent (note that there are alternative formulations that use a Thurstone-esque normal win probability link). Where Bradley-Terry models estimate parameters by full MLE estimation, Elo models (in an online-learning fashion) use gradient descent after every game is played to update team beliefs (Aldous, 2017).

More formally, in $G_{i;j;x}$, consider the estimated win probability Pr[$G_{i;j;x} = ijt_i, t_j, x_g$] from above (abbreviated as p_i). Then, let the outcome g_i be the indicator on i winning the game. Then, after the game is played, update to t_i^0, t_j^0 for some parameter K:

$$t_i^0 = t_i + K(g_i \quad p_i)$$
$$t_j^0 = t_j + K((1 \quad g_i) \quad (1 \quad p_i))$$

We can think of this as a gradient descent towards to the true MLE parameters t_i, t_j, where K is a parameter controlling the step size. There are also theoretical guarantees that this process will converge to the true parameters t_i, t_j as the number of games tends to 1. Specifically, if one considers the the ratings after x games have been played as $\overline{t_x}$ as a continuous-state Markov Chain, then (assuming logistic win probability), as $x \; ! \; 1$, the chain will converge to its stationary distribution (Aldous, 2017).

While convergence in the limit for Elo models is a strong guarantee, futures models need to be estimated when $x << 1$. In an entire NBA season, each team will only play 82 games, equating to just over 1200 total games. If Elo models don't converge quickly, or there is large variation in their results based on the observed sample, then they may not be good estimators of forward-looking team-strength.

With both Elo models and Bradley-Terry models, assuming a fixed strength parameter

estimate based on the previous games will lead to futures prediction models with lower variance than observed. This should not be surprising given our earlier discussion. These models would be missing a key source of variance in their eventual distributions – the uncertainty on the the team strength parameter. That is, Elo models (and other non-Bayesian approaches) ignores our earlier observation that the training data is itself the outcome of a random process. To combat this, most methodologies re-calculate the team strength parameter dynamically *within* a given simulation. That is if teams i and j have a simulated game against one another, their strength parameters are updated using the results of the simulated game as if the season was continuing. This methodology may be practically effective but it is unclear whether it is theoretically accurately capturing the variance from parameter uncertainty.

3.2.3 Microsoft TrueSkill

Microsoft's TrueSkill ratings system extends Elo models by incorporating Bayesian inference to handle uncertainties in skills (Herbrich et al., 2006), assuming that the rating t_i Normal(μ_i, σ_i). Note that TrueSkill actually estimates a player skill rating that is Normal with fixed variance, but to compare this approach with the above we aggregate the assumptions to the team level. The player-based formulation for TrueSkill allows it to be applied to multi-player and team-based games more appropriately than an Elo model (which assumes every team is the same entity each time it plays).

TrueSkill is estimated in an online-fashion (similar to Elo models) using Gaussian density filtering, where the posterior distribution from the last game is used as the prior distribution for the subsequent game (Herbrich et al., 2006). Similar to Elo, the update to skill ratings is a function of the expected outcome of that game, however the ratings adjustment is also a function of the number of observed games we've seen. Note that this is a natural consequence of a Bayesian framework – as we see more games the impact of the prior is reduced, so the uncertainty on team strength is reduced.

3.3 Proposed Methodology

3.3.1 Description

Our methodology will use the same logistic link function for win probability as Elo, but given the current set of games, will estimate posterior distributions for the team strength parameters. To get predictions over the outcomes of the rest of the games, we will sample team strengths from the posterior distribution and hold them constant in predicting future outcomes. Note that this differs from the dynamic update methodology Elo uses.

More formally, in the simplest case we'll define the prior on team strength t_i Normal(μ_i, σ), where σ HalfNormal(σ_0). In order to include the effect of HFA, we will include an additional bias term β Normal(β_0, τ). Then, the probability of team a beating team b will be $\frac{1}{1+\exp(\ (t_a \ \ t_b + \ ))}$.

Notably, we've yet to define any of $\mu_i, \sigma_0, \beta_0, \tau$. Recall that because we are using probabilistic programming, we can flexibly iterate through di□erent implementations for defining these parameters. Moreover, we could change the shape/distributional structure easily from the base model.

For μ_i, we might consider any of the following (among others):

1. Provide an non-informative prior ($8i, \mu_i = 0$)
2. Provide an informative prior using an estimate of team strength from previous seasons, potentially regressed back towards zero. This would be similar to an Empirical Bayes approach.
3. Use a hierarchical model, such that μ_i Normal(μ^0, σ^0).

For σ_0 and τ, we can change the information in the prior by increasing/decreasing the parameter. For β_0 we can consider similar approach as μ_i, choosing a non-informative prior ($\beta_0 = 0$) or informative prior estimated from prior data.

3.3.2 Implementation

```
with pm.Model() as model:
    # home court advantage
    hfa = pm.Normal('hfa', mu=prior_hfa, sd=0.7)

    # prior for standard deviation of team strength
    strength_sd = pm.HalfNormal('strength_sd', sd=1)

    # prior distribution on team strength
    strength = pm.Normal('strength', mu=prior_strength,
                         sd=strength_sd, shape=num_teams)

    # logistic win probability
    strength_diff = tt.dot(matchups, strength) + hfa
    p_win = pm.Deterministic('p_win', 1 / (1 + tt.exp(-strength_diff)))

    # observed data
    outcomes = pm.Bernoulli('outcomes', p=p_win, observed=results)

    # sample from posterior
    trace = pm.sample(2000, tune=1000, cores=8, target_accept=0.95)
```

Once again, the brevity and clarity of building Bayesian models in probabilistic programming is apparent. Each assumption made earlier is clearly modularized (ie. to change the assumption on home court advantage, the only place the practitioner needs to ad-

just the program is in the variable hfa). The sampling methodology (which uses NUTS) is abstracted in pm.sample() as discussed earlier – trace now holds samples from the posterior distribution.

The simplicity of this code should not be taken for granted. Implementing a similar model without using a probabilistic programming language would require explicit declarations for the likelihood of the prior distributions, full specification of a sampling algorithm and separate convergence diagnostic/evaluation frameworks. Beyond not taking advantage of e□cient trace-based computation (leading to slower sampling times), it would take more time to initially develop, has more potential for bugs and is harder to change in the future. Using a probabilistic programming language here makes it feasible for someone with little knowledge of Bayesian statistics to create this model (consider that nothing in the above required an explicit understanding of anything beyond a prior distribution) and for someone with little vector-based computer science knowledge to create an e□cient sampling mechanism for an arbitrary model.

Recall that we care about $p(R)$, which can be obtained by simulating game results with the posterior samples.

```
with model:
    # Calculate skill differences for non-played games using posterior samples
    ppc = pm.sample_posterior_predictive(trace, samples=10000,
                                         var_names = ['hfa', 'strength'])

    # Extract p_win values for non-played games R
    p_win_test = 1 / (1 + tt.exp(-(tt.dot(R,
                      ppc['strength'].T) + ppc['hfa'].T)))
```

This now gives the posterior estimates for the win probability of each game – and from this the actual results can be estimated via Monte Carlo simulation from this distribution.

```
    posterior_seasons = np.random.binomial(n=1, p=p_win_new.eval()).
```

This now provides $p(R)$, as desired (a single sample from posterior_seasons is a sample from the posterior predictive distribution over the results of all games).

3.4 Evaluation and Discussion

The above implementation was applied on the 2018-2019 NBA season, as if the day was January 1st 2019 (roughly 35 games played per team in observation and 47 games played per team remaining in the season). In order to compare this methodology with the Elo methodology, the Elo model will be built as specified:

- Elo ratings will be assumed to be centered at zero, with a HFA bias of 0.2 added to a logistic win probability function.

- The initial Elo rating for each team will be calculated by MLE on the previous season's data, regressed back to 0 by multiplying the rating by ρ.
- Elo will be continuously re-fit after every game using the formulation described above, with K = 0.05.

Further, $\mu_i, \sigma_0, \beta_0, \tau$ are specified follows:

- μ_i will be the same value as the initial Elo rating from above (ie informative).
- σ_0 will be 0.5 and τ will be 0.7 – so as to be relatively non-informative.
- β_0 will be the estimated MLE value from the previous season (similar to μ_i).

Beyond differences in the results, the methodologies differ in sampling efficiency. The Elo model must be evaluated sequentially, since within each "simulation," the Elo rating has to be updated based on the result of the fake game. Thus even with multi-processing of each simulation, samples cannot be drawn in a vectorized fashion. This contrasts with the Bayesian approach – because the strengths are sampled from the posterior distribution and held constant, the probability of victory for every game can be computed simultaneously in each "simulation". Of course, obtaining the posterior samples requires computation, but probabilistic programming implementations are designed to efficiently do this computation. Empirical results suggest that including sampling the Bayesian methodology is 4-5 times faster than the Elo methodology.

While the Elo methodology doesn't create a true posterior distribution (since it is not Bayesian), we will compare the simulated distribution of end-of-season ratings with the posterior distributions implied by our methodology. These choices were made to as closely resemble our chosen implementation, so that meaningful differences in outputs (and the distributions) are likely a result of the difference in methodology for incorporating variance as opposed to parameters/formulation. It is important to note that the Elo methodology's "distribution" is not a probability distribution because there is no probabilistic assumption on the team strength parameter (there is no sample space or probability measure). This is a relevant theoretical difference between the two methodologies since we may want the ability to make probabilistic statements about parameters (and only the Bayesian method would support that). That said, visually inspecting these differences may still be informative to understand where the methodologies differ.

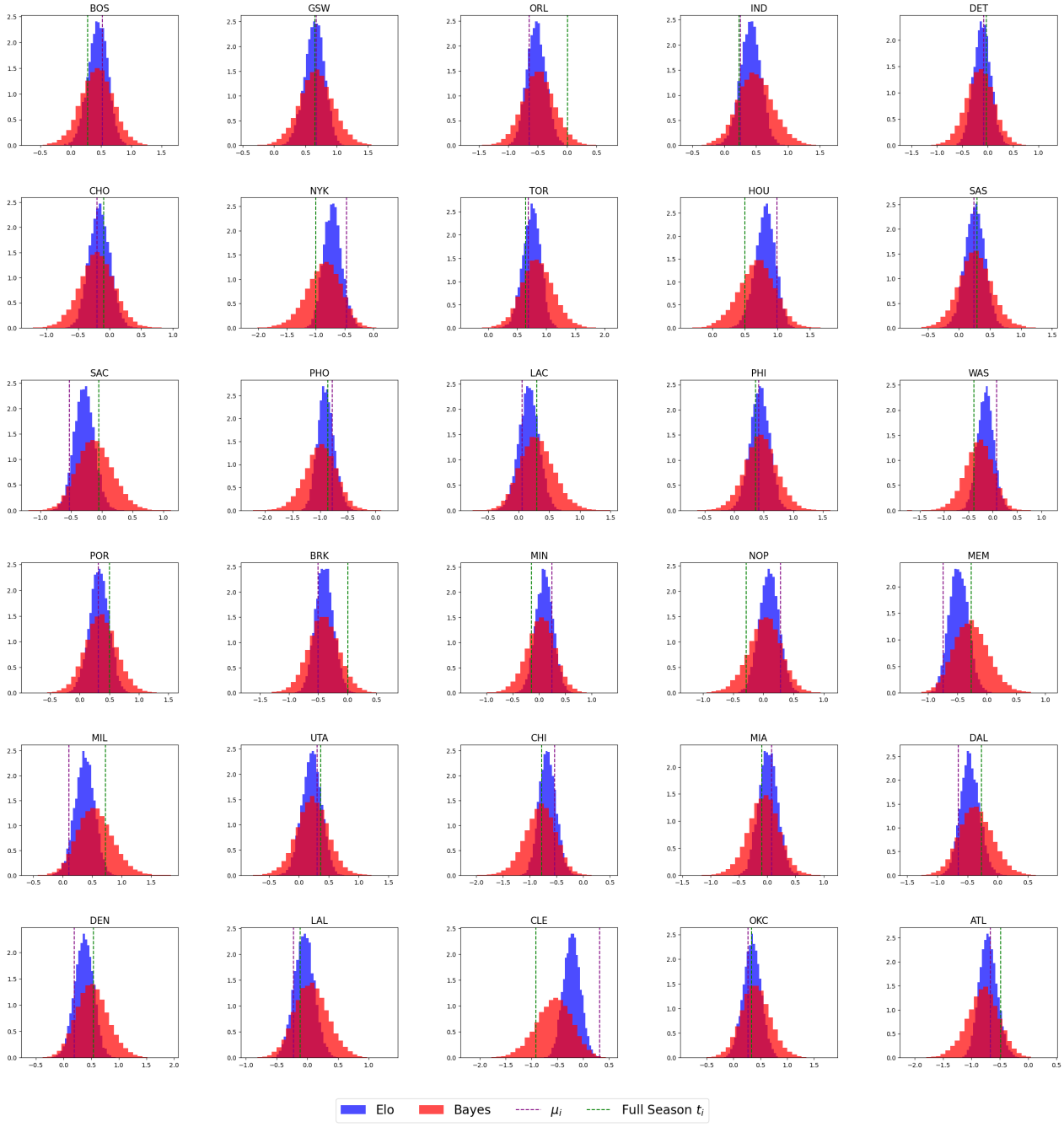


Figure 1: Comparison of t_i Posterior Distributions, 2019 NBA (35 games)

The figure above shows the output distributions for, μ_i plotted in purple and the final full-season MLE estimate for t_i is in green. There are a few high-level observations we can make. Universally, the Bayesian methodology has fatter tails – recall that this is expected because of the additional uncertainty using a prior distribution provides. For a

significant number of teams, the two distributions have a similar shape/center. For these teams, the Bayesian methodology will not lead to notable di□erences in average prediction but will predict a higher likelihood of extreme (or wing) outcomes. In other cases, the two distributions do not share the same center. We can see that in the cases where they diverge, the center for the Bayesian method is often closer to the full season t_i than the Elo methodology. Specifically, we can look at the case of CLE. At the conclusion of the 2018 season, LeBron James left the Cavaliers, and so the 2019 CLE team was among the worst in the league. The Bayesian model picks this up more quickly, because the prior distribution for CLE incorporates uncertainty on its strength – whereas the Elo model is forced to start from μ_i and update as it sees evidence that CLE is not very good.

More generally, when the full-season t_i rating significantly diverges from the initial μ_i, the Bayesian model is often quicker to pick up on the di□erence. This may be counterintuitive, since the use of a prior distribution often makes Bayesian techniques *less* reactive to new information than traditional statistical techniques. Because the Elo model is trained in an online fashion, it has to start from the starting μ_i, which may make it slower to adapt to new information. However, we note that the changing of the hyperparameters K and σ_0 could change this relationship. In fact, consider a slight alteration to the existing model that uses $K = 0.1$.

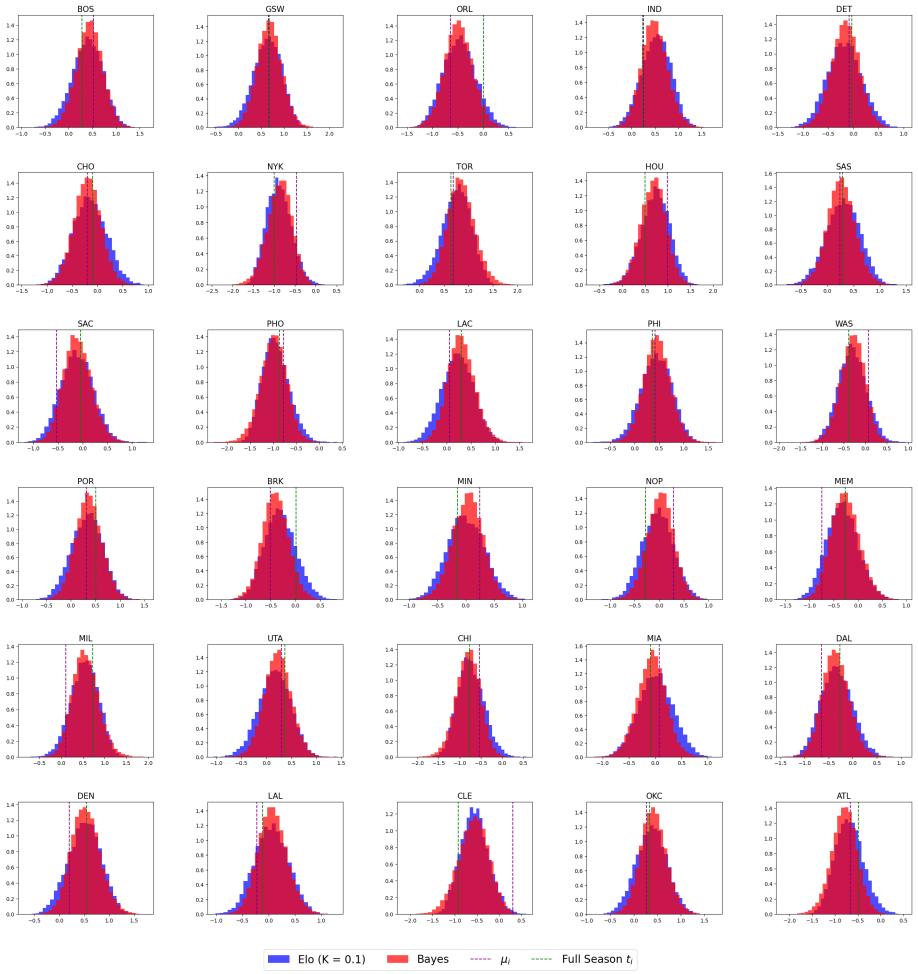


Figure 2: Comparison of t_i Posterior Distributions, 2019 NBA (35 games)

Under these settings, the resulting Elo distributions are practically indistinguishable from the Bayesian posterior distributions. Taking this a step further, we can compare the average standard deviation in the resultant distributions for di□erent parameter settings for K and σ_0 in Tables 1 and 2.

K	Value
0.01	0.031
0.05	0.159
0.1	0.316
0.2	0.610
1	2.337

Table 1: Avg Std Dev of Simulated t_i

σ_0	Value
0.1	0.1921
0.25	0.2651
0.5	0.2817
1.5	0.2876
5	0.2882

Table 2: Avg Std Dev of $p(t_i|x)$

The relationship between the Elo and Bayesian models becomes more clear. The outputted variance of the Elo model is extremely (and roughly linearly) dependent on the learning rate K. In the context of the discussion on robustness, the outputs of the Elo-based model is sensitive to the hyperparameter K, which may be undesirable. For settings where the appropriate K is not clear, this places the onus on the practitioner to either select an appropriate K, perhaps using cross-validation or other techniques. Moreover, it may mean that K needs to be re-chosen whenever augmentations or iterations on the model are made. This might further exacerbate the e□ciency di□erence between the two techniques, as performing cross-validation techniques across di□erent values of K requires repeated re-sampling.

This contrasts with the Bayesian methodology, where setting informative priors like $\sigma_0 = 0.1$ can reduce variance, but setting relatively non-informative priors lead to the same result. This makes this strategy much more robust, and far easier to iterate with (because one can safely set a non-informative prior without worrying about dramatically altering results).

To further verify this, we can compare the posterior predictive distribution of wins between the Bayesian model and Elo models with di□erent parameters of K. This is important because it will verify that the impact of di□ering posterior strength parameters permeate to the outcome of interest, R.

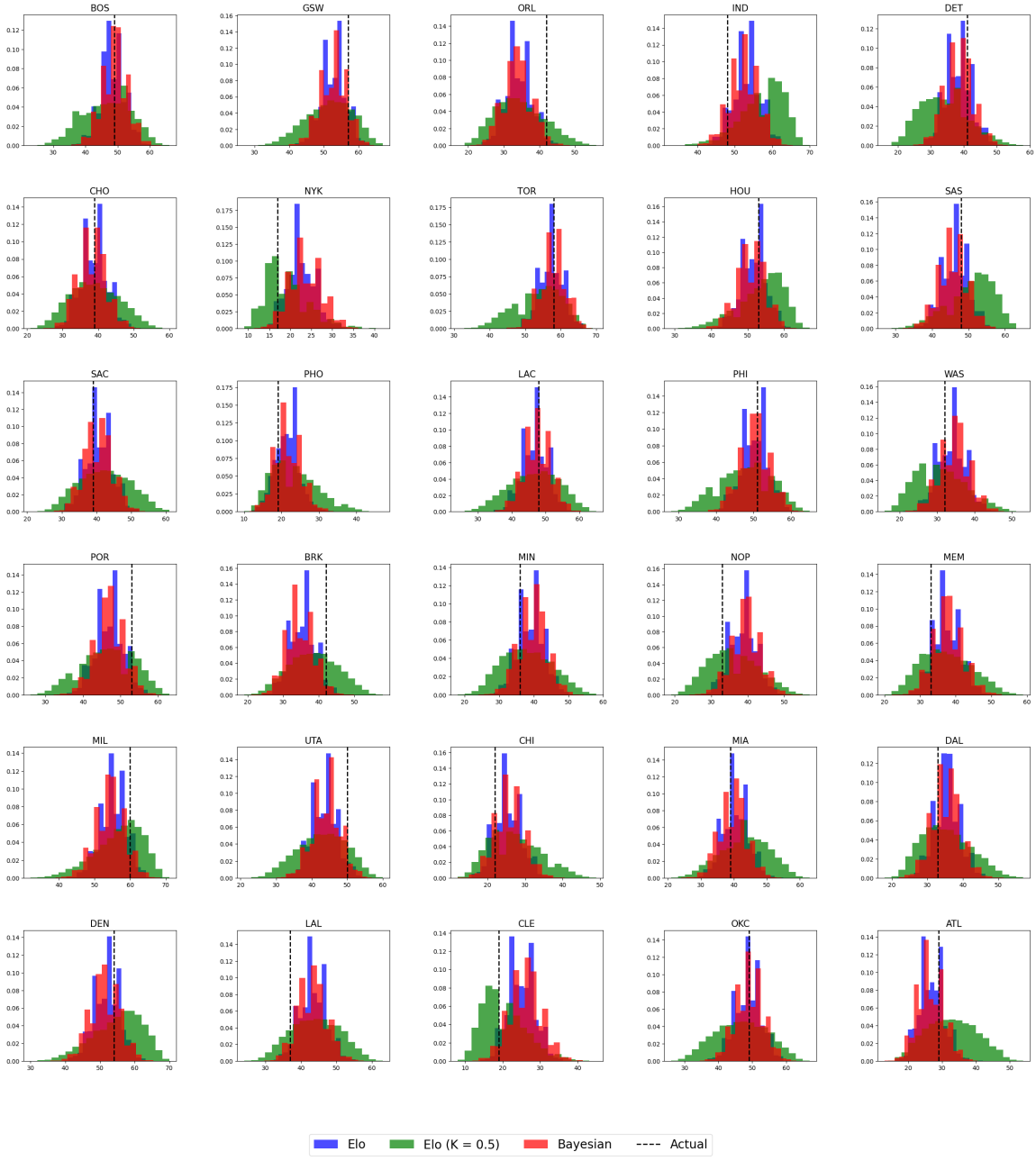


Figure 3: Comparison of Posterior Predictive Win Distributions, 2019 NBA

As expected, when K is specified correctly, the posterior predictive win distributions for the two methodologies are close to each other. However, when K is specified to a di□erent value (in this case 0.5) the variance in the resulting posterior predictive distribution is clearly di□erent – and predictions using this altered distribution may be inaccurate, as it will over-predict wing outcomes and under-predict central outcomes.

3.5 Extensions

To further highlight the flexibility that probabilistic programming provides, here are a few extensions of the simple Bayesian model that could be further pursued.

3.5.1 Hierarchical Variance

This model assumes that the variance on team skill is a shared parameter across all teams (ie every team has the same prior variance), an assumption that is fair given that teams have played roughly the same number of games at all points of the season. However, extensions of this model might relax that assumption and use σ_i HalfNormal(σ_0), defining the variance in a hierarchical fashion.

3.5.2 Non Stationarity

Both the Bayesian and Elo models assume that team strength is stationarity. Extensions of the model may relax this assumption by allowing the team strength distribution to "drift" over time.

Consider an alternative formulation where team strength starts at an initial true strength t_i and then drifts according to a Gaussian random walk. The probabilistic program could easily be refactored to support this addition as follows:

```
with pm.Model() as model:
    hfa = pm.Normal('hfa', mu=prior_hfa, sd=0.7)
    strength_sd = pm.HalfNormal('strength_sd', sd=0.25)
    initial_strength = pm.Normal('initial_strength', mu=prior_strength,
                                 sd=strength_sd, shape=num_teams)

    # Define the drift as a Gaussian random walk
    drift_sd = pm.HalfNormal('drift_sd', sd=0.1)
    strength = pm.GaussianRandomWalk('strength', sigma=drift_sd,
                         shape = (num_games, num_teams)) + initial_strength

    strength_diff = tt.dot(matchups, strength[game_time_indices].T) + hfa
```

In practice this assumption might pose computational problems (the posterior being sampled from is more difficult to estimate) – but the flexibility of probabilistic programming languages means that the random walk assumption could be adapted to a hierarchical model for different time periods, or an auto-regressive model simply by changing the distributional primitive used within `strength`. Alternatively because probabilistic programming abstracts the sampling process, different sampling algorithms/methods can be easily experimented with, including variational inference.

Conclusion

First, we highlighted the importance of Bayesian models as tools for quantifying uncertainty and modelling distributional outcomes. These models have been relatively unused because designing Bayesian models not only required a high-level statistical background, but also adequate computer science knowledge to e□ciently implement them. We then established how probabilistic programming languages are used to simplify, abstract and automate Bayesian inference. We demonstrated that probabilistic programming not only makes the code itself cleaner, but can be built to leverage the unique computational properties of the stochastic inference problem to make the code more e□cient compared to generic implementations.

Then, using the probabilistic programming language PyMC3, we explored the sports futures modelling problem, implementing a Bayesian approach to contrast with accepted industry standards. We noted that the Bayesian model had four distinct advantages over Elo-based simulation approaches:

1. Sampling from the Bayesian approach is more e□cient since it does not require sequential recalculation.
2. The Bayesian approach enables direct probability statements on team strength since it assumes a probability distribution on the latent strength parameter.
3. The Bayesian approach is more robust to hyperparameter settings. While there are settings for the learning rate K that make both methodologies virtually identical, we showed that both the "posterior" and "posterior-predictive" distributions for the Elo model can be significantly biased by the choice of K.
4. The Bayesian (and more accurately a probabilistic programming) approach enables easy extensions of the base model.

We hope that this thesis inspires further exploration into Bayesian models, specifically in sports analytics contexts where they may be particularly useful and robust compared to current state of the art.

BIBLIOGRAPHY

Aldous, D. (2017). Elo ratings and the sports model: A neglected topic in applied probability? *Statistical Science*, 32(4):616–629.

Bishop, C. M. (1994). Mixture density networks.

Burton, C., Stubbs, S., and Onyisi, P. (2021). Mixture density network estimation of continuous variable maximum likelihood using discrete training samples. *The European Physical Journal C*, 81(7).

Dahlqvist, F., Silva, A., and Kozen, D. (2020). *Semantics of Probabilistic Programming: A Gentle Introduction*, page 1–42. Cambridge University Press.

De Raedt, L. and Kimmig, A. (2015). Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47.

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2021). *Bayesian Data Analysis*. Chapman and Hall/CRC, 3rd edition.

Hamilton, I., Tawn, N., and Firth, D. (2023). The many routes to the ubiquitous bradley-terry model.

Herbrich, R., Minka, T., and Graepel, T. (2006). Trueskill™: A bayesian skill rating system. In Schölkopf, B., Platt, J., and Ho□man, T., editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press.

Ho□man, M. D. and Gelman, A. (2011). The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo.

Neal, R. M. (2003). Slice sampling. *The Annals of Statistics*, 31(3):705–741.

Roy, D. M. (2016). Probabilistic programming. https://simons.berkeley.edu/sites/default/files/docs/5675/talkprintversion.pdf. University of Toronto. Accessed: 2024-03-24.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*.

Thurstone, L. L. (1927). The method of paired comparisons for social values. *J. Abnorm. Soc. Psychol.*, 21(4):384–400.

Tolpin, D., van de Meent, J.-W., Yang, H., and Wood, F. (2016). Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2016, New York, NY, USA. Association for Computing Machinery.

van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. (2021). An introduction to probabilistic programming.

Xing, E. P. (2015). Variational inference ii. https://www.cs.cmu.edu/~epxing/Class/10708-15/notes/10708_scribe_lecture13.pdf. Accessed: 2024.

Yang, L., Yeh, Y.-T., Goodman, N. D., and Hanrahan, P. (2013). Incrementalizing mcmc in probabilistic programs through tracing and slicing.

Zanco, D. G. d. P., Szczecinski, L., Kuhn, E. V., and Seara, R. (2024). Stochastic analysis of the elo rating algorithm in round-robin tournaments. *Digital Signal Processing*, 145.