Gaussian Process for Time-To-Event Analysis

by **Ying Pan Lau**
UID: 3035230109
Supervisor: Dr. Xu, JinFeng

Master of Statistics
STAT8002 Project
Final Report

at
The University of Hong Kong
May 2022

Abstract

Time-To-Event analysis, or called Survival Analysis has been developing for decades, the famous of which are still the Cox Proportional Hazard model or the Accelerated failure time (AFT) model, which were developed in the last century. With the increasing demand of analysis in different fields, a modern model with better performance would be great to have. Due to the upsurge of Gaussian process, this study investigates the potential of Gaussian process in survival model by leveraging it to model complex relationship between survival time and covariates. Several log-logistic AFT models with or without Gaussian process were implemented, and comparisons in inference methods and evaluation metrics were also done. Though the analysis showed that the Gaussian process might not be better than the AFT model on some occasions, rooms of improvement had been discovered that, with proper investigation and specification, it should be capable of outperforming the traditional model due to its flexibility, and the capability of injecting prior knowledge.

Contents

List of Figures

List of Tables

List of Abbreviations

AFT	Accelerated Failure Time
GP	Gaussian Process
HMC	Hamiltonian Monte Carlo
LLAFT	Log-Logistic AFT
LLGP	Log-Logistic Gaussian Process AFT
LLCGP	Log-Logistic Chained Gaussian Process AFT
Matern52	Matern Kernel with its v being 5/2
MCMC	Markov Chain Monte Carlo
NUTS	No-U-Turn Sampler
RBF	Radial Basis Function
VI	Variational Inference

1 Introduction

1.1 Background

In some studies, especially in the fields of medicine, manufacturing, or marketing, time to an event, or called survival time is the event of interest. For example, it can be applied to clinical analysis, predictive maintenance, and churn prediction. If the event occurred in all individuals, many methods of analysis would be applicable. However, it is often to lose track of individuals, not seeing the events happening at the end of study, or observing the individuals experiencing another mutually exclusive event, the phenomenon of which is called right censoring. Thus, their survival time are unknown, while treating them as observed survival time or discarding them will underestimate the true time to event. Furthermore, time-to-event data are mostly not normally distributed but are skewed to the right with most events happening at an early stage. On the other hand, it is also possible that the event is rare such that the data are skewed to the left with most events happening at the very end or even unobserved. Existence of these features means that standard data analysis cannot be used. As a result, a method called survival analysis, also named as time-to-event analysis, or called survival analysis, is specially designed for it [7]. To give a brief idea of it, a typical survival dataset about cancer is demonstrated in Fig. 1.1.

In recent years, thanks to Moore's law, the computation power of modern computers has become much stronger. Also with the aid of Markov chain Monte–Carlo (MCMC) sampling, Bayesian statistics has entered the battle of modeling in a way that is relatively no longer computationally expensive, which could on the other hand leverage its data-centric property to inject prior knowledge into the model. There is also a Bayesian distribution called Gaussian process, which is a probability distribution over functions. Their flexibility and the power of incorporating domain knowledge of the specific problems might be worth investigating to see if they could bring in new insights into time-to-event analysis.

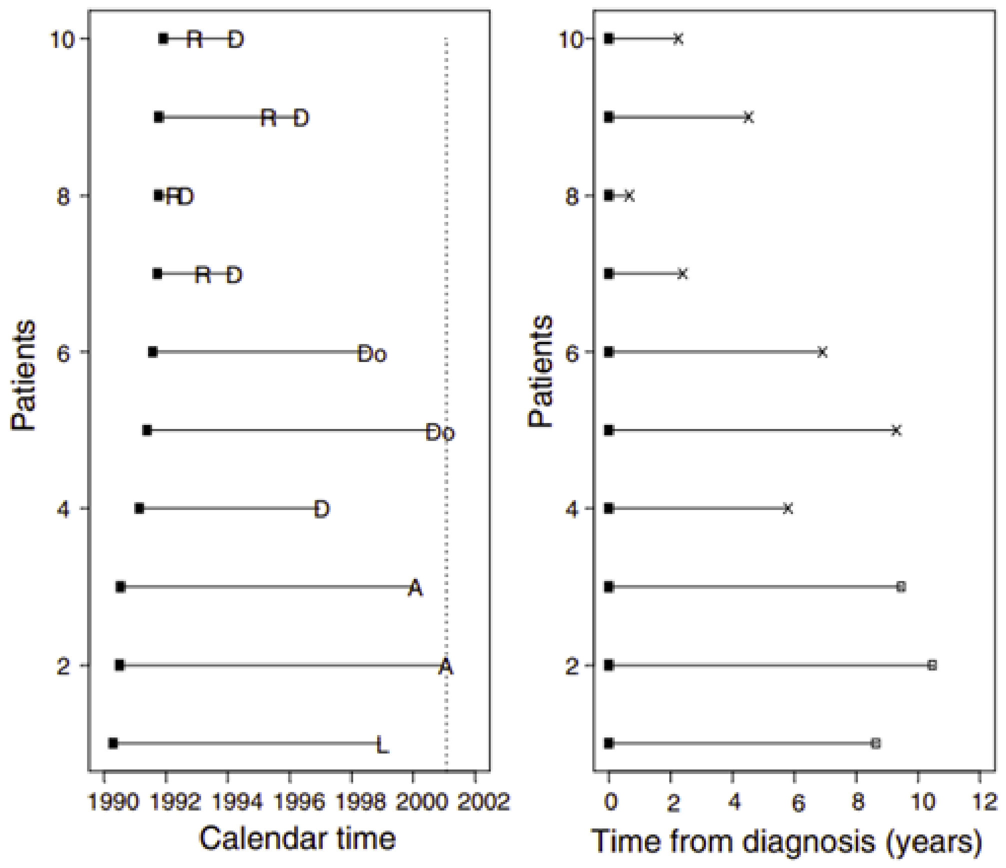


Figure 1.1: Converting calendar time in the ovarian cancer study to a survival analysis format. Dashed vertical line is the date of the last follow-up, R = relapse, D = death from ovarian cancer, Do = death from other cause, A = attended last clinic visit (alive), L = loss to follow-up, X = death, ☐ = censored [7]

1.2 Motivation

Typical survival models have strong assumption in hazard rate, or in the relationship between time and covariates. It often leads to misalignment in the actual scenario. Therefore, it should be thoroughly investigated before putting in use, or released. However, investigation takes time, and another model is still needed if the assumption fails. Here is where a non-parametric Gaussian process model could take place, to let the data model itself, and to release the assumptions and constraints.

1.3 Aim and Objectives

To explore the potential of Gaussian process survival models with its ability in exploring complex relationship between time and covariates by

- Implement a Bayesian survival model
- Implement several Gaussian process Bayesian survival models
- Explore the state-of-art inference methods for Bayesian modeling
- Compare the models with appropriate metrics

1.4 Overview

The report is structured as follows. Chapter 2 reviews related works about different survival models. Then, Chapter 3 defines and discusses several Bayesian survival models, including the Gaussian process ones, along with some discussion in traditional survival models and Gaussian process. Afterwards, Chapter 4 addresses the difficulties in computing the posterior distributions from hierarchical Bayesian models. Apart from that, Chapter 5 introduces and discusses different metrics for survival analysis and evaluates the survival models afterwards. Finally, Chapter 6 summarizes the key findings and gives ideas for future work.

2 Literature Review

Survival analysis is not a new topic, so there are lots of survival models developed. In this chapter, several typical models and advanced models based on the Gaussian process will be reviewed. In Fig. 2.1, different models are applied to the same dataset. It shows that they have their own unique survival curves. Therefore, thorough study of the models and investigation of their assumptions are needed before applying them into a specific dataset. Luckily, in the latter part of the review, the Gaussian process is brought into study, which may be able to solve this problem.

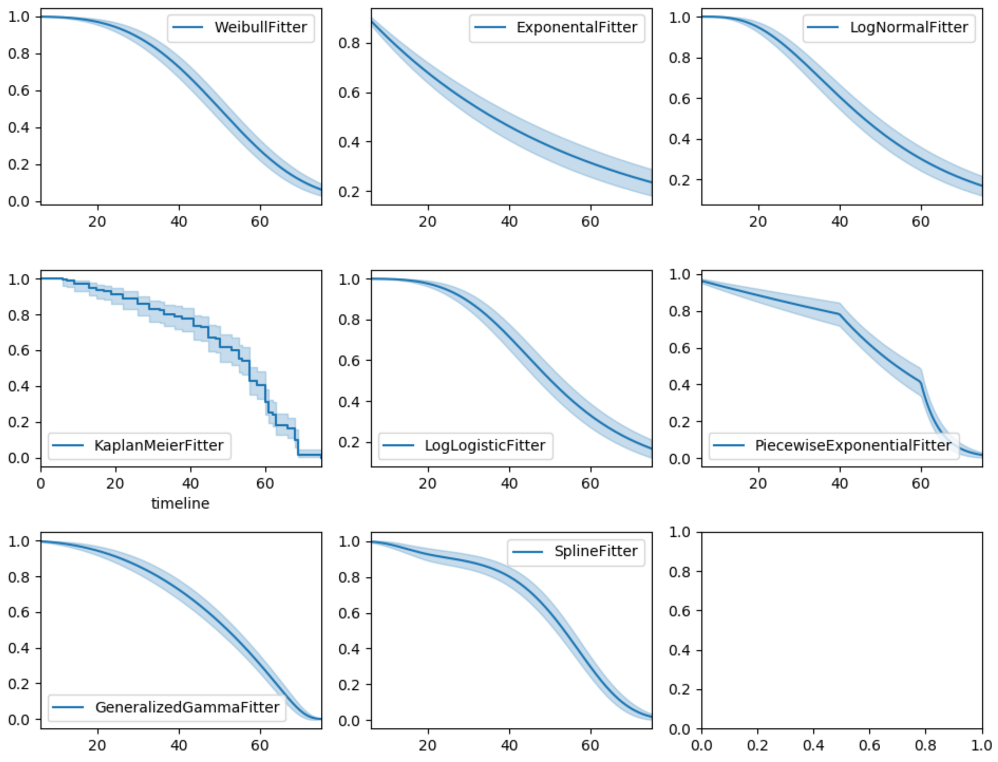


Figure 2.1: Survival model comparison

2.1 Kaplan-Meier Estimator

Kaplan-Meier Estimator is the simplest method for time-to-event analysis. It is defined as the probability of survival at a time while cutting time in many small pieces. The estimator of the survival function S(t), which means the probability that the event does not occur at least until time t, is

$$\hat{S}(t) = \prod_{i:t_{(i)} \leq t} (1 - \frac{d_i}{n_i}) \tag{2.1}$$

where t_i is the time with at least one event happened, d_i is the number of events happened at t_i, and n_i is the individuals known to have survived up to time t_i [18].

Despite it is simple as shown the above equation and in Fig. 2.2, it has few assumptions: the survival probability is the same for censored or uncensored individuals; the likelihood of the occurrence of the event of interest is the same for all the individuals no matter they are enrolled at which time; the probability of censoring is the same for any individuals; the event occurred at the defined time; And survival probability is constant within each interval [13]. In general, it is easy to violate the assumptions if there are implicit factors, lack of independence of censoring, lack of uniformity, many censoring [11] Moreover, with small sample size, the intervals will be longer, which lead to high probability in violating constant survival probability assumption. In addition, If the last observation is censored, survival function with time larger than that would be undefined. Nevertheless, it is a simple and good method to give a brief idea of the dataset on how its survival curve would look like.

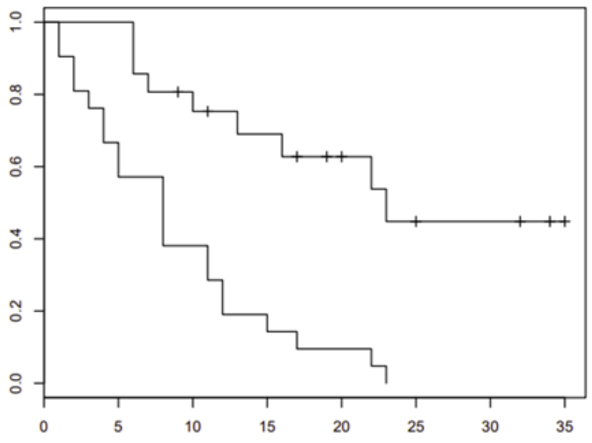


Figure 2.2: KM estimate for Gehan Data

2.2 Cox Proportional Hazard Model

Cox proportional hazard model is introduced by Cox (1972) [9], which focuses directly on the hazard function instead. Its core idea is to formulate the hazard at time t for an individual i given covariates $\boldsymbol{x_i}$ as

$$\lambda(t|\boldsymbol{x_i}) = \lambda_0(t) \exp\left(\boldsymbol{x_i}^\intercal \boldsymbol{\beta}\right) \tag{2.2}$$

Where $\lambda_0(t)$ is the baseline hazard function describing the risk of individuals with $\boldsymbol{x_i} = 0$, while $\exp\left(\boldsymbol{x_i}^\intercal \boldsymbol{\beta}\right)$ is the relative risk. The relative risk is with given covariates $\boldsymbol{x_i}$ that adjusts the baseline risk proportionally, with no interest in any time t as shown in the equation. For example, with a set of covariates $\boldsymbol{x_i}$ giving $\exp\left(\boldsymbol{x_i}^\intercal \boldsymbol{\beta}\right) = 2$, an individual has twice as much as the baseline hazard, which means that the individual is exposed as doubled baseline risk.

The idea of separating the effect of time from the effect of the covariates gives an additive model by taking logarithm [36], with

$$\log \lambda(t|\boldsymbol{x_i}) = \log \lambda_0(t) + \boldsymbol{x_i}^\intercal \boldsymbol{\beta} \tag{2.3}$$

By integrating the hazard function, the cumulative hazard is

$$\Lambda(t|\boldsymbol{x_i}) = \Lambda_0(t) \exp\left(\boldsymbol{x_i}^\intercal \boldsymbol{\beta}\right) \tag{2.4}$$

By taking exponential of negative cumulative hazard, it gives the survival function as

$$S(t|\boldsymbol{x_i}) = S_0(t) \exp\left(\boldsymbol{x_i}^\intercal \boldsymbol{\beta}\right) \tag{2.5}$$

$$S_0(t) = \exp\left(-\Lambda_0(t)\right) \tag{2.6}$$

Therefore, the covariates $\boldsymbol{x_i}$ raises the survival function to the power of relative risk.

Despite its simplicity, it makes two assumptions: the hazard functions must be proportional over the time t, and the relationship between the log hazard and covariates is linear [22].

2.3 Accelerated Failure Time Model

Accelerated Failure Time Model is defined as

$$\log T_i = \boldsymbol{x}_i^\intercal \boldsymbol{\beta} + \epsilon_i \tag{2.7}$$

where T_i is the survival time of the i-th individual, ϵ_i is a error term, with a distribution to be specified. With $\boldsymbol{x_i}$ being 0, ϵ_i is treated as baseline distribution for survival time, so on the other hand, the effect of covariates $\boldsymbol{x_i}$ is represented by $\boldsymbol{x}_i^\intercal \boldsymbol{\beta}$ as a shift on the baseline distribution. Besides, T_i is taken log because it cannot be non-negative [36]. By taking exponential of the equation, we get

$$T_i = \exp{(\boldsymbol{x}_i^\intercal \boldsymbol{\beta})} T_{0i} \tag{2.8}$$

where T_{0i} is the exponentiated error term, which is also the baseline survival time. As shown in the equation, the i-th survival time equals the baseline survival time multiplied by a non-negative effect of covariates, which is why the model is called **Accelerated** Failure Time Model.

There is another interpretation for the acceleration effect with equation

$$S(t) = S_0(\gamma t) \tag{2.9}$$

where $\gamma = \exp{(-\boldsymbol{x}_i^\intercal \boldsymbol{\beta})}$. Given $S(0) = 1$ and it is monotonically approaching to $S(\infty) = 0$, and let's say there is a person with $\gamma = 2$, according to the survival function, survival probability of that person is half of the baseline group at time t perspective. By the defined survival function, we also have

$$\lambda(t) = \gamma \lambda_0(\gamma t) \tag{2.10}$$

So if $\gamma = 2$, the person is exposed to doubled risk of the baseline group with their doubled time.

Different models can be obtained by assuming different distributions for the error term ϵ [36]. For example, if ϵ_i has an extreme value distribution with density function

$$f(\boldsymbol{x}_i^\intercal \boldsymbol{\beta}) = \exp{(\epsilon - exp(\epsilon))} \tag{2.11}$$

Then T_{0i} will have an exponential distribution, then the resulting survival model becomes an exponential model, with its hazard function being a log-linear model as

$$\log \lambda = \boldsymbol{x}_i^\intercal \boldsymbol{\beta} \tag{2.12}$$

2.4 Piecewise Exponential Model

Piecewise Exponential Model is another survival model resembles the Nelson-Aalen estimator, a asymptotically equivalent model of KM estimator [8], whose time is cut into pieces, and its hazard function is defined in each interval. It is defined as follows [36, 15].

$$\lambda(t|\boldsymbol{x_i}) = \lambda_0(t) \exp\left(\boldsymbol{x}_i^\intercal \boldsymbol{\beta}\right) \tag{2.13}$$

With its baseline hazard defined as

$$\lambda_0(t) = \lambda_j \quad \text{for } t \text{ in } [\tau_{j-1}, \tau_j) \tag{2.14}$$

To explore its benefit, a survival curve of a standard exponential model with that of a Nelson-Aalen estimate is fitted into an anonymous dataset as shown in Fig. 2.3. Compared with the non-parametric model, it shows that the exponential model performed badly with lack of fit. The Nelson-Aalen estimate indicates that there is underlying complex non-linear behavior of the true hazard rate, which cannot be captured by the exponential model [10]. However, by cutting the hazard function into pieces, that is, using a piecewise exponential model, this behavior can be captured as shown in Fig. 2.4 with a much better fit. As it may be noticed, adding more cutpoints will lead to a more perfect fitting but it will lead to overfitting. Thus, the number of cutpoints and the corresponding time range should be decided with caution.

Interestingly, this model is proven to be equivalent to a Poisson regression model [36], which allows it to be modeled in Bayesian framework with ease of implementation and computational efficiency.

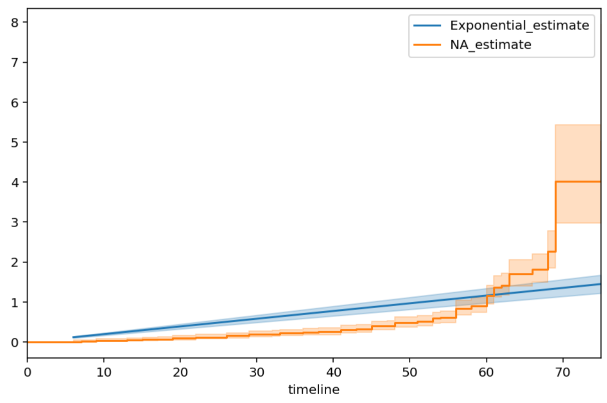


Figure 2.3: Exponential model vs Nelson-Aalen estimate [10]

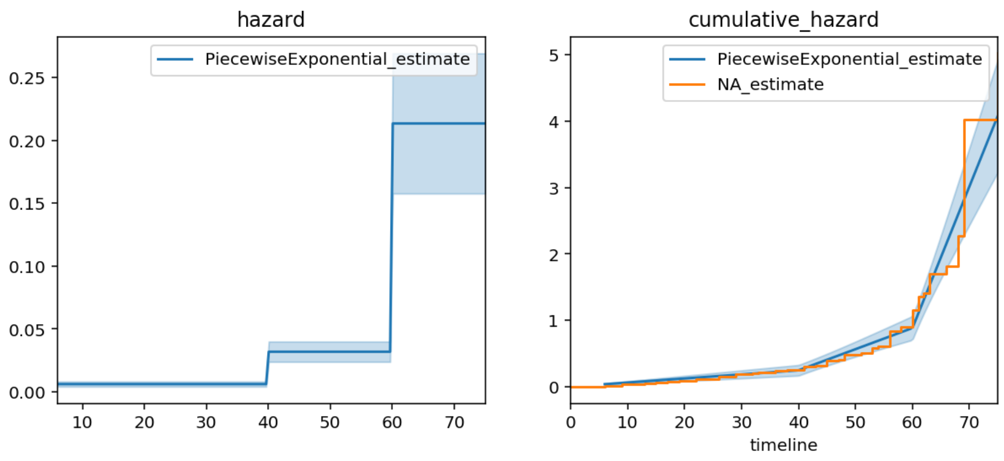


Figure 2.4: Piecewise exponential model vs NA estimate [10]

2.5 Gaussian Process Model

Gaussian Process Model is a type of survival model using Gaussian process prior, which has been developed in recent years [1, 2, 14, 38, 39]. One of the Gaussian process model is introduced by Fernandez (2016) [14], who defines the model step-by-step. He first defines a Gaussian process prior over the hazard function λ as

$$\lambda(t) = \lambda_0(t)\sigma(l(t)) \tag{2.15}$$

where $\lambda_0(t)$ is a baseline hazard function, $l(t)$ is a centered stationary Gaussian process with covariance function κ, and σ is a positive link function. The link function is chosen to be the sigmoid function $\sigma = (1 + e^{-x})^{-1}$, which is referred as the standard choice because it transforms any real number inputs into outputs in range $(0, 1)$. Then, we have the model for a data set of i.i.d. T_i without covariates as

$$l \sim \mathcal{GP}(0, \kappa), \quad \lambda(t)|l, \lambda_0(t) = \lambda_0(t)\sigma(l(t)), \quad T_i|\lambda \overset{\text{iid}}{\sim} \lambda(t)e^{-\int_0^{T_i} \lambda(s)ds} \tag{2.16}$$

The hazard function can be interpreted as a baseline hazard with a multiplicative non-parametric noise. As such, a baseline hazard can be chosen by domain experts, and then be calibrated by data with the non-parametric noise. Moreover, domain knowledge can also be injected into the covariance function κ by choosing a suitable kernel.

Finally, the survival function associated with the model has been proven to be well-defined, i.e. $S(t) \to 0$ as $t \to \infty$, given that $l(t) \sim \mathcal{GP}(0, \kappa)$ is a stationary continuous Gaussian process, $\int_0^t \kappa(s)ds = o(t)$, and it exists $K > 0$ and $\alpha > 0$ such that $\lambda_0(t) \geq Kt^{\alpha-1}$ for all $t \geq 1$. Moreover, the condition $\int_0^t \kappa(s)ds = o(t)$ is satisfied by all $\kappa(t)$ decreasing to 0, and the condition for hazard functions are satisfied by the exponential distribution [14].

The relationship between time and covariates can be modeled by the kernel of the Gaussian process prior. One way is to construct kernels for each covariate and time, and then combine them by either addition or multiplication [14]. Given time t and covariates $X \in \mathbb{R}^d$, it can be constructed such as

$$K((t_i, X_i), (t_j, X_j)) = K_0(t_i, t_j) + \sum_{k=1}^{d} K_k(X_{ik}, X_{jk}) \tag{2.17}$$

or

$$K((t_i, X_i), (t_j, X_j)) = K_0(t_i, t_j) + \sum_{k=1}^{d} X_{ik}X_{jk}K_k(t_i, t_j) \tag{2.18}$$

While the first kernel models an additive relation between time and covariates, but the second kernel models an interaction between them [14]. Other structures could also be modeled by manipulating the construction, which is also a great place for domain knowledge to be injected.

By adding the covariates, now the model is

$$l \sim \mathcal{GP}(0, K), \quad \lambda_i(t)|l, \lambda_0(t), X_i = \lambda_0(t)\sigma(l(t, X_i)), \quad T_i|\lambda_i \overset{indep}{\sim} \lambda(T_i)e^{-\int_0^{T_i} \lambda_i(s)ds} \tag{2.19}$$

To construct the kernel K, all K_k should be stationary at the end, so that the mentioned condition holds for each covariate X, which gives $S_X(t) \to 0$ as $t \to \infty$ [14].

Furthermore, not only one but also multiple Gaussian processes can be used in the survival model. What just mentioned is using one Gaussian process to model the multiplicative effect, by replacing $exp(\boldsymbol{x_i^{\intercal}\beta})$ as $\sigma(l(t, X_i))$. But it can also be extended for more parameters. For example, we can fit a log-logistic distribution into ϵ of an accelerated failure time model with Gaussian processes such that

$$T_i \sim LL(\alpha = e^{f(x_i)}, \beta = e^{g(x_i)}) \tag{2.20}$$

where $f(\boldsymbol{x}) = \mathcal{GP}(0, K_f(\boldsymbol{x}, \boldsymbol{x}^{\intercal}))$ and $g(\boldsymbol{x}) = \mathcal{GP}(0, K_g(\boldsymbol{x}, \boldsymbol{x}^{\intercal}))$. Note that the Gaussian processes are taking exponentiation because both parameters α and β have to be positive. This model releases the assumption of the shape of failure time distribution to be the same for all individuals, while allowing both skewed unimodal and exponential shaped distributions for the failure time depending on each individual [39].

The Gaussian process survival model are not restricted on single event survival analysis, but it can also be applied to competing risks. It can be solved by multiple output Gaussian process regression [2], or even deep multi-task Gaussian processes [1]. Since this paper only focuses on single event, the rest is left for readers' exploration.

3 Model

In this chapter, only Bayesian models will be constructed to model the survival time, the covariates, and the right censoring. The models will be constructed from simplest to the most advanced step-by-step, which are log-logistic accelerated failure time model (LLAFT), log-logistic Gaussian process AFT model (LLGP), and log-logistic chained Gaussian process AFT model (LLCGP). Given the definitions of survival function and hazard function stated in chapter 2, with probabilistic programming discussed in section 4.2, it allows to skip the mathematical details of the models for obtaining analytical solutions, but instead leverages the full conditional distributions and then directly obtains the inference by MCMC later stated in section 4.1.

3.1 Accelerated Failure Time Model

As mentioned in section 2.3, different AFT models can be specified by changing the prior distribution on ϵ in equation (2.7). It leads to an ease of modeling by defining $logT_i \sim \mathcal{D}(\boldsymbol{\theta})$. The associated parameters $\boldsymbol{\theta}$ are also needed to be defined, which is just to assign appropriate prior distributions to all these random variables.

In addition to the ease of modeling, it also gives the ease of interpretation with more informative results compared to the proportional hazard models described in section 2.2. For example, the results of a failure analysis of an OLED TV lifetime could be interpreted as a certain percentage change in future life expectancy on the new technology adopted compared to the original. So a product manager could be told that the TV would be expected to live, let's say, 30% longer if it adopts the new technology. On the other hand, hazard ratio is harder to be explained in layman terms. So imagining being in a meeting reporting the quantitative results to the CEO, AFT becomes a great choice.

3.1.1 Log-Logistic Accelerated Failure Time Model

Log-Logistic Accelerated Failure Time Model (LLAFT) applies a logistic prior on ϵ, it then gives its hierarchical Bayesian model architecture shown as follows.

$$\begin{aligned} loc_i &\sim Normal(0,5) \\ scale &\sim HalfNormal(5) \\ \log obs &\sim Logistic(\boldsymbol{x}^\intercal \boldsymbol{loc}, scale) \end{aligned}$$

By taking exponential transformation, we have the log-logistic distribution for observed lifetime as required

$$e^{log(obs)} \sim Loglogistic(\alpha = e^{\boldsymbol{x}^\intercal \boldsymbol{loc}}, \beta = \frac{1}{scale}) \tag{3.1}$$

A graphical representation of the model is also shown in Fig. 3.1, where log_censored_sf is a hidden function accounting the right censoring effect by multiplying the survival function of censored time into the observation likelihood.

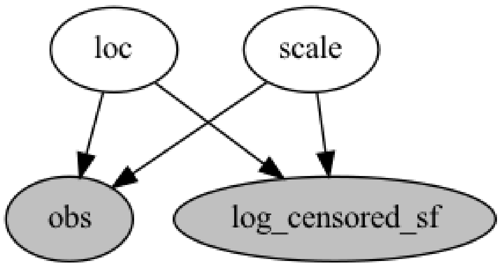


Figure 3.1: Graphical Representation of LLAFT

The Log-logistic distribution is the most commonly used AFT model. It is also chosen in this paper because it is for non-negative random variables, which is the nature of lifetime. Besides, its hazard function is non-monotonic. For example, it could be decreased at early times but be increased at later times. It resembles the log-normal distribution but with heavier tails, which makes it more robust.

Moreover, its cumulative distribution function has a closed form, which provides ease of dealing with data with censoring. Accounting the censored observation needs to be calculated with the survival function $S(T_i^+)$, which is just the complement of the cumulative distribution function $1 - P(T \leq T_i^+)$. This is an exceptionally great feature that most of the other distributions for AFT models, which includes log-normal, gamma, and inverse Gaussian distributions, does not give.

Talking about the prior distribution for loc_i being the Normal distribution, its mean being 0 is due to the prior knowledge that the multiplicative effect of the unknown covariates can either be increasing or decreasing the lifetime of an individual. If there is a strong belief or even be proven that some specific covariates will only increase or decrease the lifetime, its prior distribution could be further replaced by a half-normal distribution with negative transformation. Its variances being 5 is just for simplicity and robustness because it is large enough for logged lifetimes, which also applies to the variance of *scale*'s half-normal distribution.

Another parameter *scale* is assigned a half-normal prior distribution because it is the parameter of logistic distribution which has to be positive. If it is believed that there are often extreme cases, the distribution could be replaced by Cauchy or half-Cauchy distribution because they allows heavier tails as shown in Fig. 3.2. Cauchy distribution can also be treated as a less informative normal distribution with unknown scale due to its heavy tails property [6]. Sometimes it is a great choice for such scenarios.

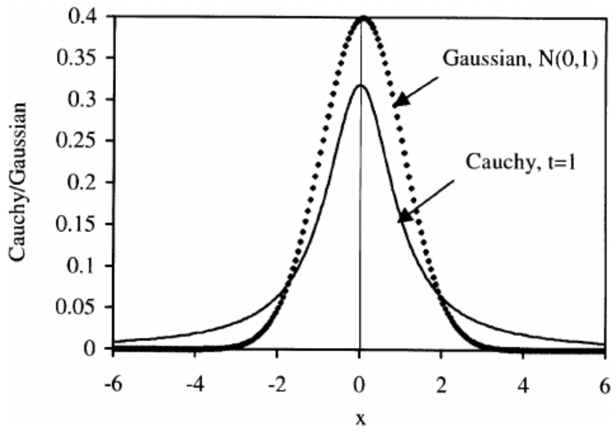


Figure 3.2: Comparison of Cauchy and Normal distribution [30]

Lastly, the covariates are added a constant one to allow constant term in *loc*, which is a standard procedure in regression model to ensure the model being unbiased, that is making the mean of the residuals being zero.

By the aid of probabilistic programming, the code for this model is as simple as in Listing 3.1

```
 def model(self, X, y=None, X_cens=None, y_cens=None, **kwargs):
     # add constant to X as a baseline, take log on y for modelling log logistic
     preprocessing_ = functools.partial(
         preprocessing, X_func=X_add_constant, y_func=jnp.log
```

```
    )

    X, y = preprocessing_(X, y)
    # apply the same transformation on censored data
    if (X_cens is not None) and (y_cens is not None):
        X_cens, y_cens = preprocessing_(X_cens, y_cens)

    loc = numpyro.sample("loc", dist.Normal(0.0, 5.0), 
sample_shape=(1, X.shape[1]))

    # Add some noise to observation
    scale = numpyro.sample("scale", dist.HalfNormal(5.0))

    # Finally, our observation model is Logistic
    y_obs = numpyro.sample("obs", dist.Logistic(loc @ X.T, 
scale), obs=y)

    # censored
    if ((X_cens is not None) and (y_cens is not None)) and (
y_cens is not None):
        constraint = 1 - dist.Logistic(loc @ X_cens.T, scale).
cdf(y_cens)
        numpyro.factor("log_censored_sf", constraint)
```

Listing 3.1: Code of LLAFT

3.2 Gaussian Process Model

The Gaussian process here is used to allow non-linear relationship between time and covariates by replacing the multiplicative effect of covariates $exp(\boldsymbol{x}_i^\intercal \boldsymbol{\beta})$. Since it is used to capture the multiplicative effect, the mean function of the Gaussian process is assumed to be 0 because it is assumed not having prior knowledge of the effect for the robustness. However, its mean function could be deeply studied to have a more appropriate one for a specific dataset. Other than the mean function being 0, a stationary kernel, which is the covariance function, must be used due to the constraint mentioned in section 2.5. With its mean being 0, selecting an appropriate kernel becomes the core of the Gaussian process because defining the kernel completely defines the process' behaviour.

There are some basic but common kernels, which are squared exponential kernel (SE) or also called radial basis function kernel (RBF), periodic kernel, and linear kernel as shown in Fig. 3.3. First of all, it is obvious that the linear kernel is non-stationary and so is not suitable. The RBF kernel is infinitely differentiable, which leads to having mean square derivatives of all orders, that is, becomes very smooth. It works well with real smooth function with no knots. However, such a strong smoothness assumption is unrealistic in the real world [41]. Meanwhile, the periodic kernel allows the model functions to repeat themselves exactly. Obviously, it is great if it is used to model periodic data, but it is rare or almost impossible in survival analysis.

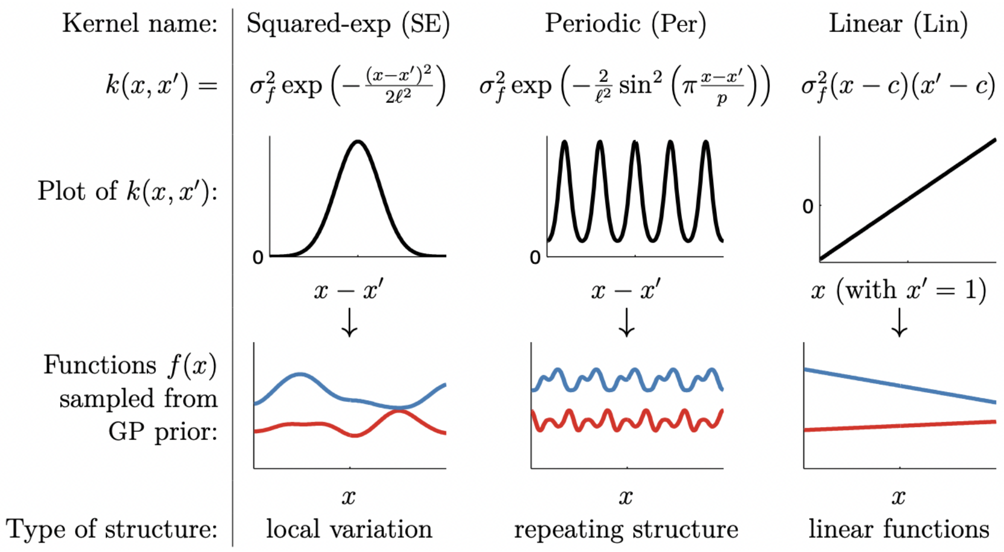


Figure 3.3: Structures of some basic kernels [12]

As stated in section 2.5, kernels could be easily combined to form a new kernel. As shown in Fig. 3.4, SE kernel plus periodic kernel give periodic function with noise, while two SE with long and short length-scale respectively give slow and fast variation. Combination is great with prior knowledge. However, for generality, the strong assumption in prior is released. As a result, this paper only adopts a type of kernel called Matern52.

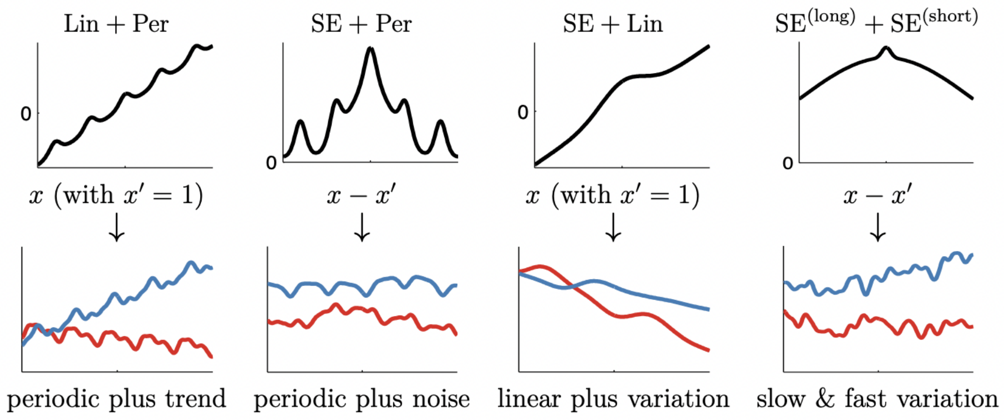


Figure 3.4: Structures of combination of some kernels [12]

Matern kernel is a kernel suggested to be used to solve the strong smoothness problem of RBF kernel, with v as its parameter to make it k-times mean square differentiable if and only if $v > k$, that is, v is controlling its smoothness. With its $v \to \infty$, it is equivalent to a RBF kernel. Besides, the kernel is very simple when v is a half-integer. There is empirical evidence that $v \geq 7/2$ is already so smooth and hard to be distinguished for noisy training examples, while $v = \frac{1}{2}$ makes the function very rough, and $v = \frac{5}{2}$ is often suggested [41]. As a result, this paper adopts Matern kernel with $v = \frac{5}{2}$, i.e. Matern52, with its equation as

$$k_{v=\frac{5}{2}}(r) = (1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2}) \exp{(-\frac{\sqrt{5}r}{l})} \tag{3.2}$$

with $r = |\boldsymbol{x} - \boldsymbol{x}^\intercal|$.

3.2.1 Log-Logistic Gaussian Process AFT Model

Log-Logistic Gaussian Process AFT Model (LLGP) builds on top of LLAFT stated in previous subsection 3.1.1 by replacing the multiplicative effect $\boldsymbol{x}_i^\intercal \boldsymbol{\beta}$ with a Gaussian process, with all of its distributions stated as follows.

$$\begin{aligned} noise\_gp &\sim HalfNormal(1) \\ rho &\sim HalfNormal(10) \\ loc &\sim \mathcal{GP}(0, K(\boldsymbol{x})) \\ scale &\sim HalfNormal(5) \\ log(obs) &\sim Logistic(loc, scale) \end{aligned}$$

where K is a Matern52 kernel with scale $l = rho$. Its graphical representation is also shown in Fig. 3.5.

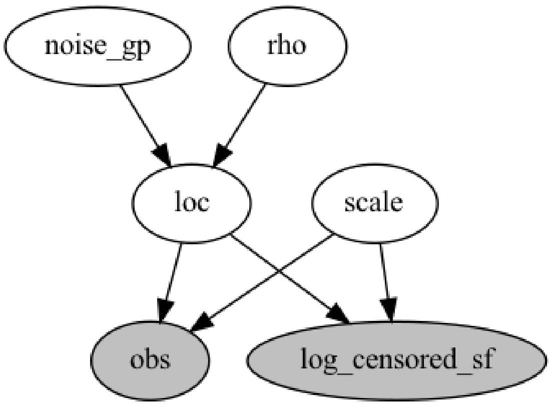


Figure 3.5: Graphical Representation of LLGP

Adding on the existing model, a Gaussian process is introduced, with its parameters *rho* and *noise_gp*. *rho* belongs to the Matern52 kernel as its length-scale, which further controls the smoothness. While *noise_gp* adds some noises into the Gaussian process to make it more robust. As a result, the original *loc* is replaced by this Gaussian process. Thus, not only the linear relationship, but also non-linear relationship between time and covariates could be captured. At last, the code for this model is in Listing 3.2.

```
        # take log on y for modeling log logistic
        preprocessing_ = functools.partial(
            preprocessing, X_func=None, y_func=jnp.log
        )

        X, y = preprocessing_(X, y)
        # apply the same transformation on censored data
        if (X_cens is not None) and (y_cens is not None):
            X_cens, y_cens = preprocessing_(X_cens, y_cens)

        if (X_cens is not None) and (y_cens is not None):
            X_ = jnp.vstack([X, X_cens])
        else:
            X_ = X

        # The parameters of the GP model
        rho = numpyro.sample("rho", dist.HalfNormal(10.0))
        noise_gp = numpyro.sample("noise_gp", dist.HalfNormal(1.0))

        if 'gp_cond' in kwargs:
            # prediction
            loc = numpyro.deterministic("loc", kwargs['gp_cond'].gp
    .mean)
        else:
            gp = build_gp(rho, noise_gp, X_)
            loc = numpyro.sample("loc", gp.numpyro_dist())

        loc1 = loc[: X.shape[0]]
        loc2 = loc[X.shape[0] :]

        # Finally, our observation model is Logistic
        scale = numpyro.sample("scale", dist.HalfNormal(5.0))

        y_obs = numpyro.sample("obs", dist.Logistic(loc=loc1, scale
    =scale), obs=y)

        # censored
        if (X_cens is not None) and (y_cens is not None):
            constraint = 1 - dist.Logistic(loc=loc2, scale=scale).
    cdf(y_cens)
            numpyro.factor("log_censored_sf", constraint)
```

Listing 3.2: Code of LLGP

3.2.2 Log-Logistic Chained Gaussian Process AFT Model

Log-Logistic Chained Gaussian Process AFT Model (LLCGP) builds on top of LLGP stated in previous subsection 3.2.1 by replacing *scale* with a Gaussian process *log_scale*. This method is called chained Gaussian process, which is discussed in section 2.5. Below are shown all of its distributions:

$$
\begin{aligned}
noise\_gp_1 &\sim HalfNormal(1)\\
noise\_gp_2 &\sim HalfNormal(1)\\
rho_1 &\sim HalfNormal(10)\\
rho_2 &\sim HalfNormal(10)\\
loc &\sim \mathcal{GP}(0, K_1(\boldsymbol{x}))\\
log\_scale &\sim \mathcal{GP}(0, K_2(\boldsymbol{x}))\\
log(obs) &\sim Logistic(loc, exp(log\_scale))
\end{aligned}
$$

where K_1 is a Matern52 kernel with scale $l = rho_1$, K_2 is also a Matern52 kernel but with scale $l = rho_2$. Its graphical representation is also shown in Fig. 3.6.

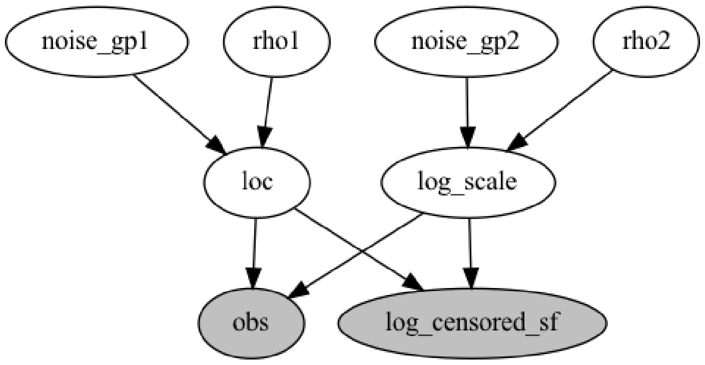


Figure 3.6: Graphical Representation of LLCGP

Adding on the existing model, one more Gaussian process is introduced, also with its parameters *rho* and *noise_gp*. As a result, the original *scale* is replaced by this Gaussian process *log_scale*, and it becomes multivariate, with each *log_scale* belonging to each individual. First of all, it is assumed to be logged so that it can take exponential to ensure the positivity assumption of the scale parameter in the logistic model. Second, assigning log_scale_i for each individual allows the flexibility for it to change under different situations. It would be great if there are indeed large changes in scale, but it may also bring in overfitting because it will be affected by noises. Lastly, the code for this model is in Listing 3.3.

```python
def model(self, X, y=None, X_cens=None, y_cens=None, **kwargs):
    # take log on y for modelling log logistic
    preprocessing_ = functools.partial(
        preprocessing, X_func=None, y_func=jnp.log
    )

    X, y = preprocessing_(X, y)
    # apply the same transformation on censored data
    if (X_cens is not None) and (y_cens is not None):
        X_cens, y_cens = preprocessing_(X_cens, y_cens)

    if (X_cens is not None) and (y_cens is not None):
        X_ = jnp.vstack([X, X_cens])
    else:
        X_ = X

    # The parameters of the GP model
    rho1 = numpyro.sample("rho1", dist.HalfNormal(10.0))
    noise_gp1 = numpyro.sample("noise_gp1", dist.HalfNormal(1.0))

    if 'gp_cond1' in kwargs:
        # prediction
        loc = numpyro.deterministic("loc", kwargs['gp_cond1'].gp.mean)
    else:
        gp1 = build_gp(rho1, noise_gp1, X_)
        loc = numpyro.sample("loc", gp1.numpyro_dist())

    # second gp
    rho2 = numpyro.sample("rho2", dist.HalfNormal(10.0))
    noise_gp2 = numpyro.sample("noise_gp2", dist.HalfNormal(1.0))

    if 'gp_cond2' in kwargs:
        # prediction
        log_scale = numpyro.deterministic("log_scale", kwargs['gp_cond2'].gp.mean)
    else:
        gp2 = build_gp(rho2, noise_gp2, X_)
        log_scale = numpyro.sample("log_scale", gp2.numpyro_dist())

        # This parameter has shape (num_data ,)
        loc1 = loc[: X.shape[0]]
        loc2 = loc[X.shape[0] :]

        # It will take exponential to ensure positivity
        log_scale1 = log_scale[: X.shape[0]]
        log_scale2 = log_scale[X.shape[0] :]

        # Finally, our observation model is Logistic
        y_obs = numpyro.sample(
            "obs", dist.Logistic(loc=loc1, scale=jnp.exp(log_scale1
)), obs=y
        )

        # censored
        if (X_cens is not None) and (y_cens is not None):
            constraint = 1 - dist.Logistic(loc=loc2, scale=jnp.exp(
log_scale2)).cdf(y_cens)
            numpyro.factor("log_censored_sf", constraint)
```

Listing 3.3: Code of LLCGP

4 Inference

In the Bayesian approach, all unknown quantities, including the predictions and parameters of a model, are random variables associated with probability distributions. **Inference** is the process to compute the posterior distribution over these quantities, conditioning on the observed data [33]. The posterior distribution is calculated by the famous Bayes' rule

$$p(\boldsymbol{\theta}|D) = \frac{p(\boldsymbol{\theta})p(D|\boldsymbol{\theta})}{p(D)} \tag{4.1}$$

The main difficulty immediately comes out in the denominator because it is needed to normalize the posterior probability, but it requires to solve a integral in high-dimensional spaces:

$$p(D) = \int p(D|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \tag{4.2}$$

Apart from that, the posterior marginal probability also becomes a issue:

$$p(\theta_i|D) = \int p(\theta_i, \boldsymbol{\theta}_{-i}|D)d\boldsymbol{\theta}_{-i} \tag{4.3}$$

To conclude, integration is the heart of doing Bayesian inference. However, for most of the time, it is difficult or even impossible to get an analytical solution for the integrations required in a probabilistic model. Here comes the approximate inference algorithms, with most of which competing and trading off speed, accuracy, simplicity, and generality. The methods include MAP estimation, grid approximation, Laplace approximation, variational inference (VI), Monte Carlo approximation, and Markov chain Monte Carlo (MCMC).

VI and MCMC are the most popular and state-of-art methods for posterior inference based on the criteria just mentioned. The former uses an optimization approach to approximate an intractable probability distribution $p(\boldsymbol{\theta}|D)$ with a tractable distribution $q(\boldsymbol{\theta})$ by minimizing the discrepancy D between the distributions [33]:

$$q^* = \underset{q \in Q}{\operatorname{argmin}}\, D(q, p) \tag{4.4}$$

where Q is some tractable family of distributions. This method is biased because it is restricted to a specific function form $q \in Q$, while MCMC is asymptotically unbiased. On the other hand, VI is much faster than MCMC, which can also be speeded up by distributed computing. When there is so much data to handle with time constraints such as dealing with big data everyday, VI wins. But when the model is dealing with small but expensive data, which is most of the case in survival data such as clinical data, MCMC wins. So MCMC is adopted and discussed in this paper, the rest methods are left for readers' exploration.

4.1 Markov Chain Monte Carlo

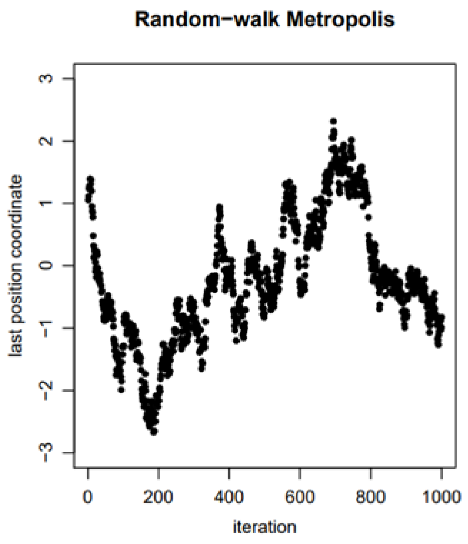
Markov Chain Monte Carlo is an inference method using sampling to approximate the desired quantities, including the predictions and parameters of a model. Its methodology is told by its name, which combines Markov chain and Monte Carlo approximation. Basically it draws samples along the Markov chain to perform Monte Carlo integration with respect to target posterior [33].

Many algorithms have been developed since a few decades ago, the most famous of which are Metropolis Hastings (MH) algorithm, random-walk Metropolis (RMH) algorithm, Gibbs sampling, Hamiltonian Monte Carlo (HMC), and No-U-Turn Sampler (NUTS).

MH algorithm and Gibbs sampling are typical, easy to understand and implement, but they rely on random search, which lead to poor performance in high-dimensional space. Luckily, there is another method called HMC, which takes gradient information to guide the moves instead of walking randomly by utilizing Hamiltonian mechanics, with NUTS being its successor.

4.1.1 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a MCMC method that uses the derivatives of the density function being sampled to generate efficient transitions spanning the posterior [6]. As shown in Fig. 4.1, it performs much better than the other usual method with a much lower standard deviation in coordinate samples. However, three hyperparameters, which are the number of steps, the step size, and the covariance, are required to be specified. With these hyperparameters being poorly selected, the performance will be largely dropped.



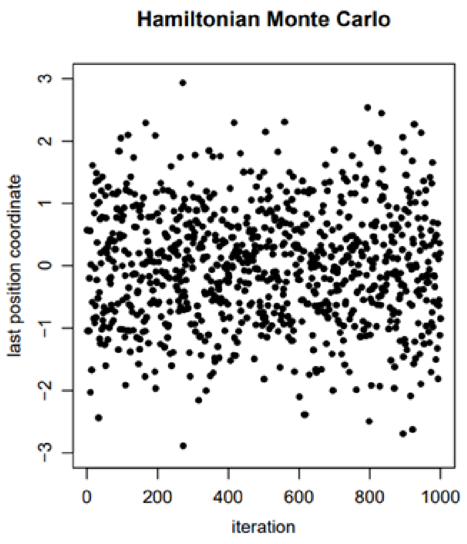


Figure 4.1: Values for the variable with largest standard deviation for the 100-dimensional example, from a random-walk Metropolis run and an HMC run with L = 150. To match computation time, 150 updates were counted as one iteration for random-walk Metropolis [34]

4.1.2 No-U-Turn Sampler

HMC's performance depends strongly on the tuning parameters. Luckily, step size and covariance have been researched to have the best candidates, while there is also a method built on top of HMC, which is called **No U-turn sampler** (NUTS), to choose the number of steps automatically. It is a way of constructing trajectories to avoid going back, while still satisfying detailed balance. To ensure detailed balance is satisfied, NUTS runs the dynamics both forward and backward in time [25]. As shown in Fig. 4.2, NUTS is now the state-of-art sampling method for Bayesian modeling. Therefore, it is adopted to be the only inference method in this paper.

4.1.3 Burn-in

MCMC leverages the Markov chain, and thus it needs an initial amount of iterations to obtain the stationary distribution. Such an amount of iterations to be discarded before it starts to draw samples from the posterior distribution are called burn-in. It can be seen in Fig. 4.3, where the trace climbed up at

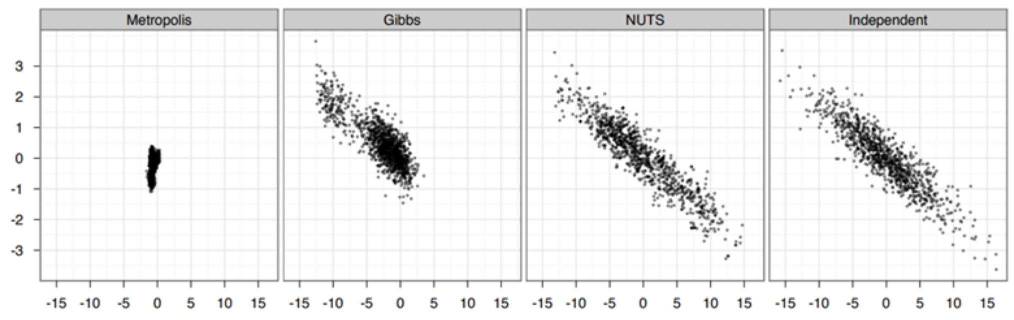


Figure 4.2: Samples generated by random-walk Metropolis, Gibbs sampling, and NUTS. The plots compare 1,000 independent draws from a highly correlated 250-dimensional distribution (right) with 1,000,000 samples (thinned to 1,000 samples for display) generated by random-walk Metropolis (left), 1,000,000 samples (thinned to 1,000 samples for display) generated by Gibbs sampling (second from left), and 1,000 samples generated by NUTS (second from right). Only the first two dimensions are shown here [25]

the burn-in period which is shaded at first half, then somehow became stationary. Without sufficient burn-in samples to be discarded, the posterior quantity drawn by calculating mean will be largely biased with large variances. Moreover, the samples after burn-in might be stationary with large variances. As a result, a sufficient amount of burn-in and samples are needed to eliminate bias and variance.

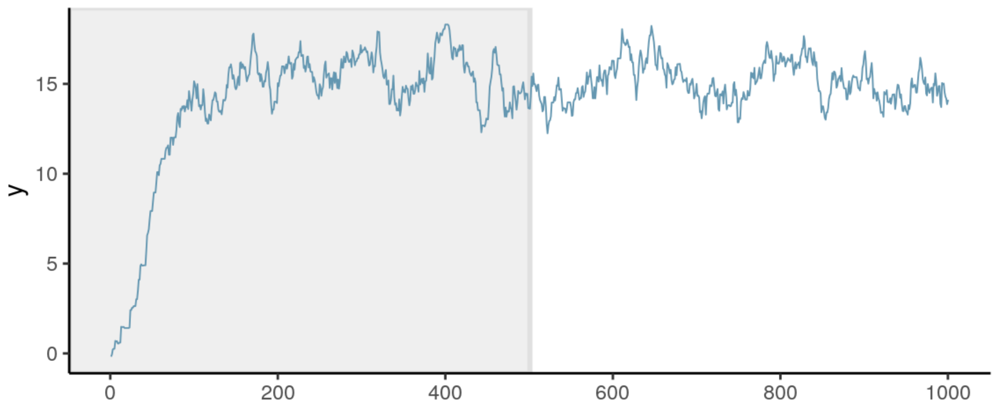


Figure 4.3: Sampling trace with burn-in [6]

4.2 Probabilistic Programming

Construction of the Bayesian model, NUTS algorithm, and most of the associated data manipulation are difficult to implement. Luckily, Bayesian modelling is popular nowadays, and MCMC method is generic enough to be implemented for whatever the model is. As a result, probabilistic programming is developed, and there are existing libraries in different programming languages. For example, Stan [6] in C++ and R , Turing.jl [17] in Julia, and PyMC [37], Pyro [4] with its successor NumPyro [35] in Python. Finally, to work in Python, NumPyro is chosen because it is tested to outperform PyMC and Pyro in speed [26, 29].

5 Evaluation

5.1 Evaluation Metrics

A model has to be evaluated before putting it to use. Its performance is mostly evaluated by the accuracy of prediction, while the accuracy should take account of the problem setting. So different metrics may be the best in different use cases. As a result, two mostly used metrics with different pros and cons are going to be introduced.

5.1.1 Concordance Index

Concordance Index is first introduced by Harrell et al. (1982) [21], which has been the most popular metric for survival analysis since then. It is defined as

$$C = \frac{\sum_{i,j} \delta_j 1_{T_j > T_i} 1_{f_j > f_i}}{\sum_{i,j} \delta_j 1_{T_j > T_i}}. \tag{5.1}$$

where i, j represents a pair of individual i and j, δ_j is the indicator variable of the event observation of individual j, T_i is the observed survival time of individual i, f_i is the predicted survival time of individual i, and $1_{f_j > f_i}$ is the indicator function for a subset with condition $f_j > f_i$. In words, it is reporting the proportion of the number of concordant pairs to the total number of possible evaluation pairs. As in the equation, the C-index is ranged from 0 to 1, with 1 being the best, while 0.5 being the random guess. With the C-index being lower than 0.5, it means that the model is so worse that the modeler should instead toss a coin to predict.

This metric is intuitive and easy to interpret. However, it only measures the discrimination, that is, the order of prediction, but not the calibration, which is, the predictions themselves. For example, the C-index for a model perfectly predicting patients death time is 1, while the C-index for a model wrongly predicting patients death time, which is off by an order of magnitude, but preserving the right order of death, is also 1. Sometimes it is enough, but sometimes calibration is also needed to be accessed for a comprehensive performance analysis [27].

Apart from that, this metric does not evaluate the prediction accuracy based on the time horizon of the prediction, which sometimes is an important measure in predictive analytic. For instance, in churn analysis, a one-year time frame might be the most concerned time horizon for a company running yearly subscription services. Assuming that, a model doing the best in 1-year time horizon but doing the worse in 10-year time horizon would still be chosen, yet the C-index is useless in this decision making [27].

5.1.2 Brier Score

Brier Score is a score function to measure the accuracy of probabilistic predictions, which is equivalent to the mean squared error to the predicted probabilities, and is defined as follows.

$$BS = \frac{1}{N}\sum_{t=1}^{N}(f_t - o_t)^2 \tag{5.2}$$

where f_t is the probability of the event at time t being predicted, o_t is the actual outcome of the event at time t, and N is the number of forecasts. As from the equation, it is ranged between 0 and 1, and the lower the Brier score is, the better the predictions are.

This metric measures not only discrimination, but also calibration. Moreover, a specified time horizon is taken into account. However, it is inadequate for very rare or very frequent events, since small changes that are significant in such events cannot be sufficiently discriminated [3].

To provide a deeper insight, Brier score can be decomposed into three terms as

$$BS = \frac{1}{N}\sum_{k=1}^{K} n_k(f_k - \bar{o}_k)^2 - \frac{1}{N}\sum_{k=1}^{K} n_k(\bar{o}_k - \bar{o})^2 + \bar{o}(1-\bar{o}) \tag{5.3}$$

where K is the number of unique forecasts, $\bar{o} = \sum_{t=1}^{N}\frac{o_t}{N}$ is the observed base rate for the event to occur, n_k is the number of forecasts within the same category, and $\bar{o}_k$ is the observed frequency given f_k. In brief, the first term is a reliability term measuring how close the forecast probabilities are to the true probabilities, which is known as calibration, while the second term is a resolution term measuring how much the conditional probabilities differ from average, and the last term is a uncertainty term measuring the inherent uncertainty in the outcomes of the event [32, 24].

With it being applied to survival analysis with right censoring, it is further modified to

$$BS(t) = \frac{1}{N}\sum_{t=1}^{N}\left(\frac{(0-\hat{S}(t|\boldsymbol{x_i}))^2}{\hat{G}(T_i^-)}1_{T_i\le t,\delta_i=1} + \frac{(1-\hat{S}(t|\boldsymbol{x_i}))^2}{\hat{G}(t)}1_{T_i>t}\right) \tag{5.4}$$

where $\hat{G}(t) = P(C > t)$ is the survival function of the censoring time estimated by the Kaplan-Meier estimator, and δ_i is the indicator variable for individual i, with it being one as observing the event [20].

5.2 Model Evaluation

In this subsection, there models were applied into a synthetic dataset [10] with a short description listed in Listing 5.1. Subsamples with different sample sizes will be fitted. Afterwards, C-index and brier score with 5 time points, which are minimum, 25-th quantile, 50-th quantile, 75-th quantile, and maximum of the sub-sampled lifetimes.

```
Size: (200,5)
Examples:
          var1      var2      var3         T  E
     0.595170  1.143472  1.571079  14.785479  1
     0.209325  0.184677  0.356980   7.336734  1
     0.693919  0.071893  0.557960   5.271527  1
     0.443804  1.364646  0.374221  11.684168  1
     1.613324  0.125566  1.921325   7.637764  1
     ...
```

Listing 5.1: Short description of the synthetic dataset

As discussed in section 4.1.3, sufficient amount of burn-in and samples are needed. Since total cases in this dataset is 200, and there are quite a lot of parameters needed to be sampled as defined in chapter 3, the number of burn-in and number of samples are chosen to be 5000 each. The evaluation result is shown in Table 5.1. According to the table, as the number of samples increases, the runtime of Gaussian process models largely increases. The LLCGP model's runtime is even much larger than the LLGP model, with its worst time being 419.4 seconds, while the runtime of the LLAFT model was nearly not changed, with the worst time being 4.2 seconds. As what is discussed in section 2.5, the kernel calculates the distance between each pair of covariates. It is also proven that inverse of the $n \times n$ matrix is needed, which leads to a computational difficulty with time complexity being $O(n^3)$ [23]. The LLCGP model is the slowest because it involves two Gaussian processes which need to be sampled sequentially due to the nature of MCMC method.

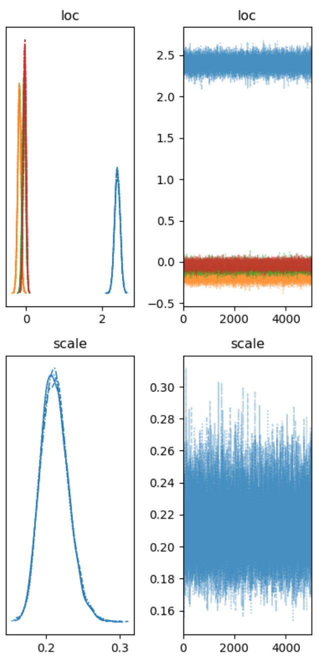
In the view of performance, it is observed that LLAFT outperformed others when the sample size is small, while the sample size increased, the performance gaps were closed and then the Gaussian process models became better according to C-index. It may be due to the complexity of the model, and the flexibility of Gaussian processes. The former requires more samples to be fitted into the model to avoid overfitting, while the latter is too flexible that it allows the observed value to appear in any spaces, which leads to underfitting. Underfitting would be improved as sample size increases because observed value with its covariates will act as knots that restrict the output of Gaussian processes [19, 40]. In addition, it is also observed that the brier score of the simple LLAFT model outperformed the others in any sample sizes or time specified. The increasing trend in C-index with respect to sample size and underperforming brier score indicate that the Gaussian process models are improving in discrimination but bad in calibration.

model	size	runtime(s)	C-index	BS$_{min}$	BS$_{q25}$	BS$_{q50}$	BS$_{q75}$	BS$_{max}$
LLAFT	20	4.0	0.780	0.052	0.089	0.195	0.099	0.103
LLGP	20	6.6	0.718	0.036	0.159	0.285	0.108	0.108
LLCGP	20	19.8	0.740	0.040	0.193	0.297	0.107	0.108
LLAFT	50	4.1	0.668	0.056	0.190	0.182	0.052	0.029
LLGP	50	61.7	0.654	0.055	0.243	0.257	0.062	0.031
LLCGP	50	84.3	0.670	0.060	0.241	0.265	0.062	0.031
LLAFT	100	4.2	0.595	0.029	0.209	0.187	0.029	0.015
LLGP	100	135.6	0.603	0.030	0.297	0.242	0.032	0.016
LLCGP	100	419.4	0.606	0.030	0.249	0.243	0.032	0.016

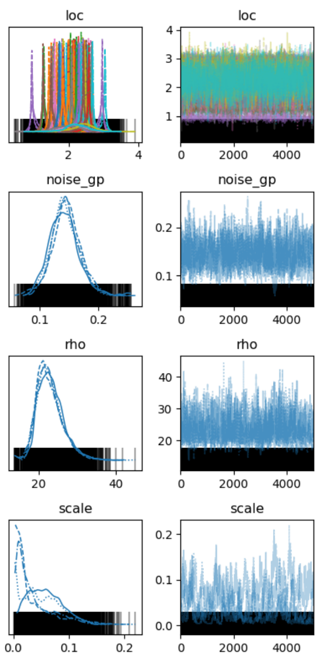
Table 5.1: Results of applying different models with different sample size with size being sample size, BS being brier score

Trace plots in Fig. 5.1 shows that parameters of the LLAFT model are stationary, while parameters of the Gaussian process models are almost stationary with a certain amount of divergences which are shown as black bars at the bottom. It indicates that more burn-ins are required. In addition, it may also be solved by reparameterization [6], which could dramatically increase effective sample size for the same number of iterations or even make programs that would not converge well behaved.

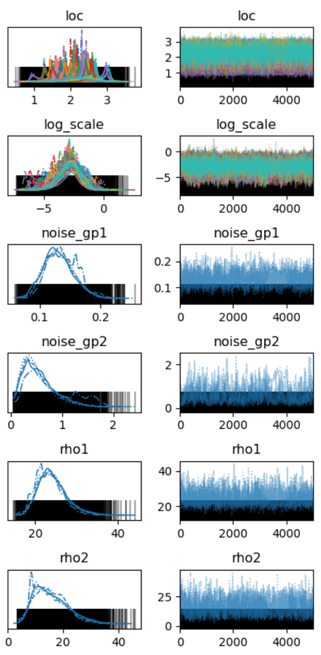
The modeled survival functions of the models with sample size as 100 are shown in Fig. 5.2. As shown in Fig. 5.2(a), the slopes of LLAFT's survival curves under different covariates changed smoothly. However, the slopes of Gaussian



(a) LLAFT



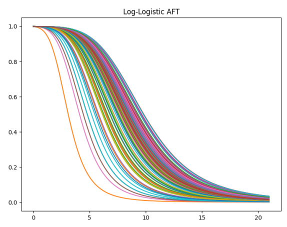
(b) LLGP



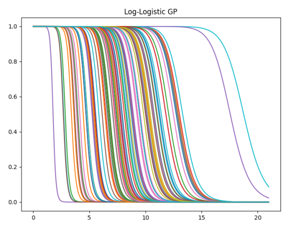
(c) LLCGP

Figure 5.1: MCMC trace plots of the models at sample size = 100

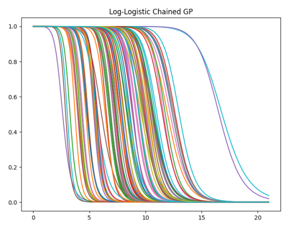
process models' survival curves were similarly steep in the dense region, while some of them were slightly changed at the end. It shows overfitting that survival curves changed from 1 to 0 within a short time in the dense region. It not only solves the underfitting problem stated in last paragraph, but also makes it too much to the stage of overfitting. It is suspected that *noise_gp* defined for the Gaussian processes in section 3.2.1 and 3.2.2, which represents the diagonal variances added into the covariance matrix, are too large that suppresses the covariance between covariates, such that the dependency between covariates are not large enough to affect each other much.



(a) LLAFT



(b) LLGP



(c) LLCGP

Figure 5.2: Survival curves of the models at sample size = 100

6 Conclusions and Future Work

This research aimed to explore the potential of using the Gaussian process in survival models. Based on the practical implementation and quantitative evaluation, this succeeded in building Bayesian log-logistic AFT models with Gaussian processes, which have a simple closed form survival function to deal with censoring, as stated in section 3.1.1. The LLAFT model is able to deal with covariates and censoring, while the Gaussian process models were developed to model the complex relationship between time and covariates but there is some fine-tuning to be done to maximize its capability, as stated in section 5.2.

It is also found in section 5.2 that the Gaussian process is hard to be leveraged due to its cubic time complexity, which makes it computationally expensive to be solved. Though it indeed has potential to perform well, it takes enormous time to do so, which often could not be done in time in a fast-paced work environment. After trading off time and sample size, it may turn out to be worse than a simple model which has a much bigger throughput.

It is also found that the Gaussian process is completely defined by its kernel when its mean function is 0, which is often the case, as stated in section 3.2. This property is both good and bad. Bad thing is that it is very complicated but also unrestrained, which makes it difficult to make the proper choice for a given scenario. Good thing is that once knowledge about it has been comprehensively developed, with a proper kernel construction and optimization of its parameters, prior knowledge could be well injected into the model to make better predictions.

It is worth mentioning the flexibility of the Gaussian process, which allows prior knowledge to be injected not only into the parameters, but also into the model itself during the model construction stage by constructing its kernel. Its kernel is so flexible that it could fit for almost every data, including non-numeric data, such as text, or image [12]. Apart from that, combination of different kernels is allowed as stated in section 3.2, which includes multiplication and addition, integrating the properties of different types of kernel. This feature allows experienced researchers to build almost whatever they think without being worried about how to inject their ideas into a mathematical model.

This research built the models on top of the AFT model, while it would also be interesting to explore the proportional hazard model. As stated in section 2.4, the piecewise proportional hazard model is somehow a Poisson regression model. Therefore, it is suggested to work in this direction to explore the potential of using the Gaussian process in the proportional hazard model, modeling a more complex relationship between covariates and hazard.

Focusing on the AFT model developed, there are rooms to improve. Finding a better kernel would be a great direction since there are so many possibilities in kernel construction. Refining the prior distribution would also be a great choice such as replacing the constant variances with a bigger value, or replacing the normal distribution with a Cauchy distribution to make the priors less informative. On the contrary, if making a generic model for every scenario is not the focus, much more informative priors could be assigned with a deeper understanding in specific situations, allowing faster convergence, more accurate posteriors. Reparameterization stated in section 5.2 is worth mentioning again because it makes the model more well-formed by rewriting the expectation to make the corresponding MCMC estimate unbiased, differentiable, and scalable [28]. Moreover, Variational Inference discussed in section 4 and scalable Gaussian process have been rapidly developing over the last decade [23, 16, 31, 5]. Replacing MCMC with VI, or speeding up the Gaussian process by different implementations are worth exploring too. Lastly, with the Gaussian process being a function of functions, and its output being the input in the next step, it is so indirect and complicated that most of the time struggling in developing is just to figure out what happened and how it would behave. Thus, it would be great to develop a graphical representation of the underlying Gaussian process with its evolution.

Bibliography

[1] A. M. Alaa and M. van der Schaar. “Deep multi-task gaussian processes for survival analysis with competing risks”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 2326–2334.

[2] J. E. Barrett and A. C. Coolen. “Gaussian process regression for survival data with competing risks”. In: *arXiv preprint arXiv:1312.1591* (2013).

[3] R. Benedetti. “Scoring rules for forecast verification”. In: *Monthly Weather Review* 138.1 (2010), pp. 203–211.

[4] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. “Pyro: Deep universal probabilistic programming”. In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 973–978.

[5] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. “Variational inference: A review for statisticians”. In: *Journal of the American statistical Association* 112.518 (2017), pp. 859–877.

[6] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. “Stan: A probabilistic programming language”. In: *Journal of statistical software* 76.1 (2017).

[7] T. G. Clark, M. J. Bradburn, S. B. Love, and D. G. Altman. “Survival analysis part I: basic concepts and first analyses”. In: *British journal of cancer* 89.2 (2003), pp. 232–238.

[8] E. Colosimo, F. v. Ferreira, M. Oliveira, and C. Sousa. “Empirical comparisons between Kaplan-Meier and Nelson-Aalen survival function estimators”. In: *Journal of Statistical Computation and Simulation* 72.4 (2002), pp. 299–308.

[9] D. R. Cox. “Regression models and life-tables”. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 34.2 (1972), pp. 187–202.

[10] C. Davidson-Pilon. “lifelines: survival analysis in Python”. In: *Journal of Open Source Software* 4.40 (2019), p. 1317.

[11] *Does your data violate Kaplan-Meier assumptions?* Web Page. 2022. URL: https://www.quality-control-plan.com/StatGuide/kaplan_ass_viol.htm.

[12] D. Duvenaud. “Automatic model construction with Gaussian processes”. PhD thesis. University of Cambridge, 2014.

[13] I. Etikan, K. Bukirova, and M. Yuvali. “Choosing statistical tests for survival analysis”. In: *Biom. Biostat. Int. J* 7 (2018), pp. 477–481.

[14] T. Fernández, N. Rivera, and Y. W. Teh. “Gaussian processes for survival analysis”. In: *Advances in Neural Information Processing Systems* 29 (2016).

[15] M. Friedman. “Piecewise exponential models for survival data with covariates”. In: *The Annals of Statistics* 10.1 (1982), pp. 101–113.

[16] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson. “Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration”. In: *Advances in neural information processing systems* 31 (2018).

[17] H. Ge, K. Xu, and Z. Ghahramani. “Turing: a language for flexible probabilistic inference”. In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 2018, pp. 1682–1690. URL: http://proceedings.mlr.press/v84/ge18b.html.

[18] M. K. Goel, P. Khanna, and J. Kishore. “Understanding survival analysis: Kaplan-Meier estimate”. In: *International journal of Ayurveda research* 1.4 (2010), p. 274.

[19] J. Görtler, R. Kehlbeck, and O. Deussen. “A Visual Exploration of Gaussian Processes”. In: *Distill* (2019). https://distill.pub/2019/visual-exploration-gaussian-processes. DOI: 10.23915/distill.00017. URL: http://dx.doi.org/10.23915/distill.00017.

[20] E. Graf, C. Schmoor, W. Sauerbrei, and M. Schumacher. “Assessment and comparison of prognostic classification schemes for survival data”. In: *Statistics in medicine* 18.17-18 (1999), pp. 2529–2545.

[21] F. E. Harrell, R. M. Califf, D. B. Pryor, K. L. Lee, and R. A. Rosati. “Evaluating the yield of medical tests”. In: *Jama* 247.18 (1982), pp. 2543–2546.

[22] D. Hashim and E. Weiderpass. “Cancer Survival and Survivorship”. In: *Encyclopedia of Cancer (Third Edition)*. Ed. by P. Boffetta and P. Hainaut. Third Edition. Oxford: Academic Press, 2019, pp. 250–259. ISBN: 978-0-12-812485-7. DOI: https://doi.org/10.1016/B978-0-12-801238-3.65102-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128012383651024.

[23] J. Hensman, N. Fusi, and N. D. Lawrence. “Gaussian processes for big data”. In: *arXiv preprint arXiv:1309.6835* (2013).

[24] J. Hernández-Orallo, P. A. Flach, and C. F. Ramirez. “Brier curves: a new cost-based visualisation of classifier performance”. In: *Icml*. 2011.

[25] M. D. Hoffman, A. Gelman, et al. "The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo." In: *J. Mach. Learn. Res.* 15.1 (2014), pp. 1593–1623.

[26] M. Ingram. *MCMC for big datasets: Faster sampling with jax and the GPU*. 2021. URL: https://www.pymc-labs.io/blog-posts/pymc-stan-benchmark/.

[27] M. W. Kattan and T. A. Gerds. "The index of prediction accuracy: an intuitive measure useful for evaluating risk prediction models". In: *Diagnostic and prognostic research* 2.1 (2018), pp. 1–7.

[28] D. P. Kingma and M. Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[29] V. La, T. H. Nguyen, Y. Wang, J. Chodera, and D. D. L. Minh. *Benchmark of three probabilistic programming languages for Bayesian analysis of isothermal titration calorimetry data*. 2021. URL: https://www.pymc-labs.io/blog-posts/pymc-stan-benchmark/.

[30] B. Li, J. Lin, and X. Yao. "Characteristics of the optimization of creep constitutive equations". In: *Jixie Gongcheng Xuebao/Chinese Journal of Mechanical Engineering* 39 (Mar. 2003), pp. 37–39.

[31] H. Liu, Y.-S. Ong, X. Shen, and J. Cai. "When Gaussian process meets big data: A review of scalable GPs". In: *IEEE transactions on neural networks and learning systems* 31.11 (2020), pp. 4405–4423.

[32] A. H. Murphy. "A new vector partition of the probability score". In: *Journal of Applied Meteorology and Climatology* 12.4 (1973), pp. 595–600.

[33] K. Murphy. *Probabilistic machine learning: Advanced topics*.

[34] R. M. Neal et al. "MCMC using Hamiltonian dynamics". In: *Handbook of markov chain monte carlo* 2.11 (2011), p. 2.

[35] D. Phan, N. Pradhan, and M. Jankowiak. "Composable effects for flexible and accelerated probabilistic programming in NumPyro". In: *arXiv preprint arXiv:1912.11554* (2019).

[36] G. Rodríguez. *Generalized Linear Models*. Web Page. 2016. URL: https://data.princeton.edu/wws509.

[37] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. "Probabilistic programming in Python using PyMC3". In: *PeerJ Computer Science* 2 (2016), e55.

[38] A. D. Saul. "Gaussian process based approaches for survival analysis". PhD thesis. University of Sheffield, 2016.

[39] A. D. Saul, J. Hensman, A. Vehtari, and N. D. Lawrence. "Chained gaussian processes". In: *Artificial Intelligence and Statistics*. PMLR. 2016, pp. 1431–1440.

[40] Y. Shi. “Gaussian Processes, not quite for dummies”. In: *The Gradient* (2019). URL: https://thegradient.pub/machine-learning-ancient-japan/.

[41] C. K. Williams and C. E. Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006.

A Python Code for Model Implementation

```python
import functools
from timeit import default_timer as timer

import numpy as np
import pandas as pd
from tinygp import kernels, GaussianProcess
from jax import numpy as jnp
from jax import scipy as jsp
from jax import random
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive
import arviz as az
import matplotlib.pyplot as plt
from lifelines.utils import concordance_index
#from sksurv.metrics import brier_score
#from sksurv.util import Surv

from tte import utils
from tte.dataloader import get_sample, dataset_map


def X_add_constant(X):
    return jnp.insert(X, 0, values=1, axis=1)


def preprocessing(X, y, X_func, y_func):
    id_func = lambda x: x
    if X_func is None:
        X_func = id_func
    if y_func is None:
        y_func = id_func

    if y is not None:
        y = y_func(y)
    return X_func(X), y
```

```python


class SurvivalModel:
    _model_name = ''
    _model_shortname = ''

    def model(self, X, y=None, X_cens=None, y_cens=None):
        raise NotImplementedError

    def sf(self):
        raise NotImplementedError

    def sfs(self):
        raise NotImplementedError

    def plot_sf(self, covariates, sf):
        label = f"{self._model_name} with {covariates}"
        ax.plot(ts, sf, label=label)

    def render_model(self, X, y, X_cens, y_cens, render_distributions, render_params):
        # print model architecture
        g = numpyro.render_model(
            self.model,
            model_kwargs={
                "X": X,
                "y": y,
                "X_cens": X_cens,
                "y_cens": y_cens,
            },
            render_distributions=render_distributions,
            render_params=render_params,
        )

        g.render(
            filename=f"{utils.get_project_root()}/results/{self._model_shortname}_model{'_dists' if render_distributions else ''}",
            format="png",
            engine="dot",
        )

class LogLogisticModel(SurvivalModel):
    _model_name = 'Log-Logistic AFT'
    _model_shortname = 'll'

    def model(self, X, y=None, X_cens=None, y_cens=None, **kwargs):
        # add constant to X as a baseline, take log on y for modelling log logistic
        preprocessing_ = functools.partial(
            preprocessing, X_func=X_add_constant, y_func=jnp.log
        )

        X, y = preprocessing_(X, y)
        # apply the same transformation on censored data
        if (X_cens is not None) and (y_cens is not None):
            X_cens, y_cens = preprocessing_(X_cens, y_cens)

        loc = numpyro.sample("loc", dist.Normal(0.0, 5.0), sample_shape=(1, X.shape[1]))

        # Add some noise to observation
        scale = numpyro.sample("scale", dist.HalfNormal(5.0))

        # Finally, our observation model is Logistic
        y_obs = numpyro.sample("obs", dist.Logistic(loc @ X.T, scale), obs=y)

        # censored
        if ((X_cens is not None) and (y_cens is not None)) and (y_cens is not None):
            constraint = 1 - dist.Logistic(loc @ X_cens.T, scale).cdf(y_cens)
            numpyro.factor("log_censored_sf", constraint)

    def sf(self, ts, X, loc, scale):
        # constant term
        X_ = X_add_constant(X)
        return jsp.stats.logistic.sf((jnp.log1p(ts) - (loc @ X_.T).T) / scale)

    def sfs(self, idata, ts, X, loc=None, scale=None):
        loc = idata.posterior["loc"].mean(dim=("chain", "draw")).data
        scale = idata.posterior["scale"].mean(dim=("chain", "draw")).data
        sfs = self.sf(ts, X, loc, scale)
        return sfs

def build_gp(rho, noise_gp, X):
    gp = GaussianProcess(
        kernels.Matern52(rho),
        X,
        diag=noise_gp,
    )
    return gp

class LogLogisticGPModel(SurvivalModel):
    _model_name = 'Log-Logistic GP'
    _model_shortname = 'llgp'

    def model(self, X, y=None, X_cens=None, y_cens=None, **kwargs):
        # take log on y for modelling log logistic
        preprocessing_ = functools.partial(
            preprocessing, X_func=None, y_func=jnp.log
        )

        X, y = preprocessing_(X, y)
        # apply the same transformation on censored data
        if (X_cens is not None) and (y_cens is not None):
            X_cens, y_cens = preprocessing_(X_cens, y_cens)

        if (X_cens is not None) and (y_cens is not None):
            X_ = jnp.vstack([X, X_cens])
        else:
            X_ = X

        # The parameters of the GP model
        rho = numpyro.sample("rho", dist.HalfNormal(10.0))
        noise_gp = numpyro.sample("noise_gp", dist.HalfNormal(1.0))

        if 'gp_cond' in kwargs:
            # prediction
            loc = numpyro.deterministic("loc", kwargs['gp_cond'].gp
    .mean)
        else:
            gp = build_gp(rho, noise_gp, X_)
            loc = numpyro.sample("loc", gp.numpyro_dist())


        loc1 = loc[: X.shape[0]]
        loc2 = loc[X.shape[0] :]

        # Finally, our observation model is Logistic
        scale = numpyro.sample("scale", dist.HalfNormal(5.0))

        y_obs = numpyro.sample("obs", dist.Logistic(loc=loc1, scale
    =scale), obs=y)

        # censored
        if (X_cens is not None) and (y_cens is not None):
            constraint = 1 - dist.Logistic(loc=loc2, scale=scale).
    cdf(y_cens)
```

```python
            numpyro.factor("log_censored_sf", constraint)

    def sf(self, ts, loc, scale):
        return jsp.stats.logistic.sf((jnp.log1p(ts) - loc) / scale)

    def sfs(self, idata, ts, X=None, loc=None, scale=None):
        if loc is None:
            loc = idata.posterior["loc"].mean(dim=("chain", "draw")).data
        if scale is None:
            scale = idata.posterior["scale"].mean(dim=("chain", "draw")).data

        sfs = self.sf(ts, loc[:, jnp.newaxis], scale)
        return sfs


class LogLogisticChainedGPModel(SurvivalModel):
    _model_name = 'Log-Logistic Chained GP'
    _model_shortname = 'llcgp'

    def model(self, X, y=None, X_cens=None, y_cens=None, **kwargs):
        # take log on y for modelling log logistic
        preprocessing_ = functools.partial(
            preprocessing, X_func=None, y_func=jnp.log
        )

        X, y = preprocessing_(X, y)
        # apply the same transformation on censored data
        if (X_cens is not None) and (y_cens is not None):
            X_cens, y_cens = preprocessing_(X_cens, y_cens)

        if (X_cens is not None) and (y_cens is not None):
            X_ = jnp.vstack([X, X_cens])
        else:
            X_ = X

        # The parameters of the GP model
        rho1 = numpyro.sample("rho1", dist.HalfNormal(10.0))
        noise_gp1 = numpyro.sample("noise_gp1", dist.HalfNormal(1.0))

        if 'gp_cond1' in kwargs:
            # prediction
            loc = numpyro.deterministic("loc", kwargs['gp_cond1'].gp.mean)
        else:
            gp1 = build_gp(rho1, noise_gp1, X_)
            loc = numpyro.sample("loc", gp1.numpyro_dist())

        # second gp
        rho2 = numpyro.sample("rho2", dist.HalfNormal(10.0))
        noise_gp2 = numpyro.sample("noise_gp2", dist.HalfNormal(1.0))

        if 'gp_cond2' in kwargs:
            # prediction
            log_scale = numpyro.deterministic("log_scale", kwargs['gp_cond2'].gp.mean)
        else:
            gp2 = build_gp(rho2, noise_gp2, X_)
            log_scale = numpyro.sample("log_scale", gp2.numpyro_dist())

        # This parameter has shape (num_data,)
        loc1 = loc[: X.shape[0]]
        loc2 = loc[X.shape[0] :]

        # It will take exponential to ensure positivity
        log_scale1 = log_scale[: X.shape[0]]
        log_scale2 = log_scale[X.shape[0] :]

        # Finally, our observation model is Logistic
        y_obs = numpyro.sample(
            "obs", dist.Logistic(loc=loc1, scale=jnp.exp(log_scale1)), obs=y
        )

        # censored
        if (X_cens is not None) and (y_cens is not None):
            constraint = 1 - dist.Logistic(loc=loc2, scale=jnp.exp(log_scale2)).cdf(y_cens)
            numpyro.factor("log_censored_sf", constraint)

    def sf(self, ts, loc, scale):
        return jsp.stats.logistic.sf((jnp.log1p(ts) - loc) / jnp.exp(scale))

    def sfs(self, idata, ts, X=None, loc=None, scale=None):
        if loc is None:
            loc = idata.posterior["loc"].mean(dim=("chain", "draw")).data
        if scale is None:
            scale = idata.posterior["log_scale"].mean(dim=("chain", "draw")).data

        sfs = self.sf(ts, loc[:, jnp.newaxis], scale[:, jnp.newaxis])
        return sfs


def df2arr(df, event_col, duration_col, covariate_cols, event_val):
    # X need not to be np.nan in this stage
    X = df.loc[df[event_col] == event_val, covariate_cols].to_numpy()
    y = df.loc[df[event_col] == event_val, duration_col].to_numpy()

    X_cens = df.loc[df[event_col] != event_val, covariate_cols].to_numpy()
    y_cens = df.loc[df[event_col] != event_val, duration_col].to_numpy()
    return X, y, X_cens, y_cens

def combine_y(y, y_cens):
    return jnp.concatenate([y, y_cens])

def combine_X(X, X_cens):
    return jnp.vstack([X, X_cens])

def get_events(y, y_cens):
    return [*[1]*len(y), *[0]*len(y_cens)]


if __name__ == '__main__':
    numpyro.set_host_device_count(4)
    seed_num = 0
    rng_key = random.PRNGKey(seed_num)


    model_instance = LogLogisticModel()
    subsample = True
    subsample_size = 20
    num_warmup_mcmc = 5000
    num_samples_mcmc = 5000
    render_model = False
    only_render = False

    # cancer, rossi, synthetic, covid
    dataset_name = 'covid'
    event_col, duration_col, covariate_cols, event_val = dataset_map[dataset_name]

    df = get_sample(dataset_name)
    if subsample:
        df = df.sample(subsample_size, random_state=rng_key)

    X, y, X_cens, y_cens = df2arr(df, event_col, duration_col,
    covariate_cols, event_val)

    # for later plotting survival curve
    covariates = np.vstack([X, X_cens])

    if render_model:
        model_instance.render_model(
            X, y, X_cens, y_cens,
            render_distributions=True, render_params=True)

    if only_render:
        exit()

    # Run the MCMC
    nuts_kernel = NUTS(
        model_instance.model,
    )

    mcmc = MCMC(
        nuts_kernel,
        num_warmup=num_warmup_mcmc,
        num_samples=num_samples_mcmc,
        num_chains=4,
        progress_bar=True,
    )
    start = timer()
    mcmc.run(rng_key, X=X, y=y, X_cens=X_cens, y_cens=y_cens)
    end = timer()
    print(f"Sampling time: {end - start}s")
    mcmc.print_summary()
    samples = mcmc.get_samples()

    # MCMC sampling trace

    idata = az.from_numpyro(mcmc)

    az.plot_trace(idata, figsize=(4, 8))
    plt.tight_layout()

    # Survival Plot

    ts = np.linspace(0, y.max(), 10000)

    fig, ax = plt.subplots(figsize=(8, 6))

    sfs = model_instance.sfs(idata, ts, X=X)

    for covariate, sf in zip(covariates, sfs):
        model_instance.plot_sf(covariate, sf)

    if len(sfs) < 10:
        ax.legend()
    ax.set_title(model_instance._model_name)
    plt.show()
    exit()

    # testing
    def get_y_pred(predictions):
        mean_predictions = jnp.mean(predictions, axis=0)

        # difference between AFT and GPAFT
        if len(mean_predictions.shape) == 2:
            mean_predictions = mean_predictions[0]
        y_pred = jnp.expm1(mean_predictions)
        return y_pred

    def combine_y_result(y_true, y_pred, event_values):
        df = pd.DataFrame([y_true, y_pred]).T
        df.columns = ['true', 'pred']
        df['event'] = event_values
        return df

    X, y, X_cens, y_cens = df2arr(df, event_col, duration_col,
   covariate_cols, event_val)
    event_values = [*[1]*len(y), *[0]*len(y_cens)]
    X_ = jnp.vstack([X, X_cens])
    y_true = np.concatenate([y, y_cens])

    # build gp
    gp_cond = None
    gp_cond1 = None
    gp_cond2 = None
    loc_cond = None
    scale_cond = None
    if isinstance(model_instance, LogLogisticGPModel):
        post_loc = jnp.mean(samples['loc'], axis=0)
        post_rho = jnp.mean(samples['rho'], axis=0)
        post_noise_gp = jnp.mean(samples['noise_gp'], axis=0)
        gp = build_gp(post_rho, post_noise_gp, covariates)
        gp_cond = gp.condition(post_loc, X_)
        loc_cond = gp_cond.gp.mean
    elif isinstance(model_instance, LogLogisticChainedGPModel):
        post_loc = jnp.mean(samples['loc'], axis=0)
```

```python
        post_rho1 = jnp.mean(samples['rho1'], axis=0)
        post_noise_gp1 = jnp.mean(samples['noise_gp2'], axis=0)

        post_log_scale = jnp.mean(samples['log_scale'], axis=0)
        post_rho2 = jnp.mean(samples['rho2'], axis=0)
        post_noise_gp2 = jnp.mean(samples['noise_gp2'], axis=0)

        gp1 = build_gp(post_rho1, post_noise_gp1, covariates)
        gp_cond1 = gp1.condition(post_loc, X_)
        loc_cond = gp_cond1.gp.mean

        gp2 = build_gp(post_rho2, post_noise_gp2, covariates)
        gp_cond2 = gp2.condition(post_log_scale, X_)
        scale_cond = gp_cond2.gp.mean


    predictive = Predictive(model_instance.model, samples)
    predictions = predictive(rng_key, X=X_, gp_cond=gp_cond, gp_cond1=gp_cond1, gp_cond2=gp_cond2)["obs"]
    y_pred = get_y_pred(predictions)
    df = combine_y_result(y_true, y_pred, event_values)
    print(df)

    c_index = concordance_index(df['true'], df['pred'], df['event'])
    #survival_test = Surv.from_dataframe('event', 'true', df)

    ts = np.linspace(y_true.min()+1, y_true.max()-1, 5)
    pred_sf = model_instance.sfs(idata, ts, X=X_, loc=loc_cond, scale=scale_cond)

    #b_score = brier_score(survival_test, survival_test, pred_sf, np.linspace(y_true.min()+1, y_true.max()-1, 5))
    print(f"{c_index:.3f}")
    #print(b_score[1].round(3))
```