Active Learning for Gaussian Processes with Mixed Data

Luan Fletcher

April 2022

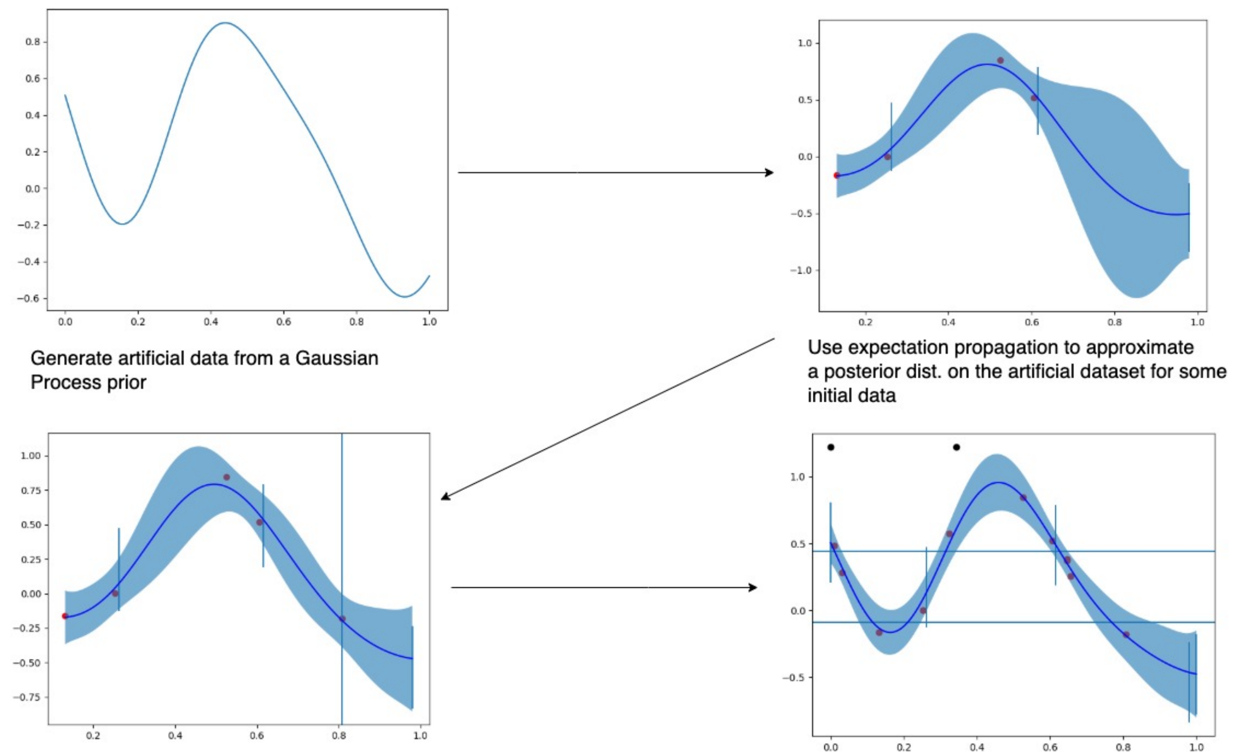
Contents

1 Introduction

Active learning (AL) is a process whereby a learning algorithm chooses particular data points at which to query an output. Instead of the learning algorithm receiving training data at random, it can choose to see training data at which it is uncertain about the output. By doing this, we can improve sample efficiency and speed up model training. This is particularly useful in contexts where the amount of training data we have available to use is limited, as AL allows us to choose training data that is maximally informative about the process we are trying to model.

AL is typically done for homogeneous datasets (datasets consisting of only one datatype). In this project, we detail a method for AL in the context of heterogeneous datasets. We make the assumption that our heterogeneous dataset is generated from some underlying latent function f which we model using a Gaussian Process prior. GP models are very flexible and can be used for a wide range of tasks such as classification, regression, preference learning etc. For this reason they are an attractive option for experimentation with novel active learning methods.

We begin by adapting a well-known method (expectation propagation) for fitting Gaussian Processes to non-Gaussian data for use with mixed data. We then derive a method for active learning where our learning algorithm can query both a point x_* and a data-type. We do this in the context of resource management, i.e. one has some budget C and costs c_i associated with querying each data-type and one wishes to maximise training efficiency within the constraints of the budget. We further generate artificial datasets on which to test our proposed method. We are not aware of any previous implementation of EP/active learning for mixed data in the context of Gaussian Processes.


Generate artificial data from a Gaussian Process prior

Use expectation propagation to approximate a posterior dist. on the artificial dataset for some initial data

Use active learning algorithm at each timestep to request a datatype at a particular point (e.g. here AL requests point data at x = 0.81)

Continue requesting data points with AL until budget is exhausted

Above we give a visualisation of the basic steps for active learning for a GP model in one dimension. We first generate our artificial dataset. We then sample some initial (mixed) training data and approximate a posterior distribution to fit these data. We then hand over the training process to the AL algorithm, which requests training data at particular input points and for particular data-types. The AL algorithm continues until our budget is exhausted.

We can give a simple illustrative motivating example for why this is an interesting problem. Imagine a lab wishes to model some output over inputs in space and time. For example, we could imagine them wishing to model soil acidity over some two dimensional plot of space. GPs are an obvious choice for modelling this problem. Assume further that they have available to them some imprecise but cheap methods for determining pH along with some expensive but precise methods. We could, for example, assume the imprecise method gives some interval of plausible pH for a given point whereas the precise method gives a low variance point estimate of the pH. The method we derive and implement in this project could be used to decide on a strategy to optimally pick the mix of precise and imprecise methods to maximize training efficiency.

2 Background

In this section we will discuss some of the concepts and tools used in the rest of the project. Notation in particular is borrowed from the excellent textbook Gaussian Processes for Machine Learning [1]

2.1 Gaussian Processes

Formally, a Gaussian Process (GP) is a collection of random indexed variables such that any finite subset of these variables has a joint Gaussian distribution. In this project, we will take the indexing set to be $\mathbb{R}^n$ for some specific n. We can thus think of a Gaussian Process as specifying a distribution over functions of $\mathbb{R}^n$. This distribution is typically used in machine learning and statistical contexts as a prior distribution on an unknown function f in the context of Bayesian inference.

A GP is completely specified by its mean function $\mu(x)$ and its covariance function $k(x_1, x_2)$. When used as a prior on a function f, we write

$$f \sim \mathcal{GP}(\mu(\cdot), k(\cdot, \cdot))$$

.

Which here means that for any set of $x_1, x_2, \dots, x_m \in \mathbb{R}^n$ we have

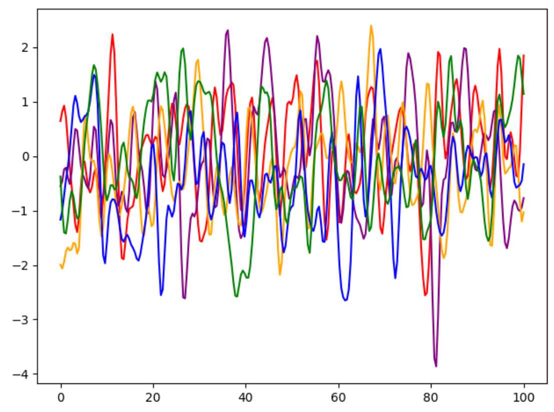
$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix} \sim \mathcal{N}(\mu(x), \mathcal{K}) \tag{1}$$

Where $\mathcal{K}$ is a covariance matrix with entry (i, j) being $k(x_i, x_j)$.

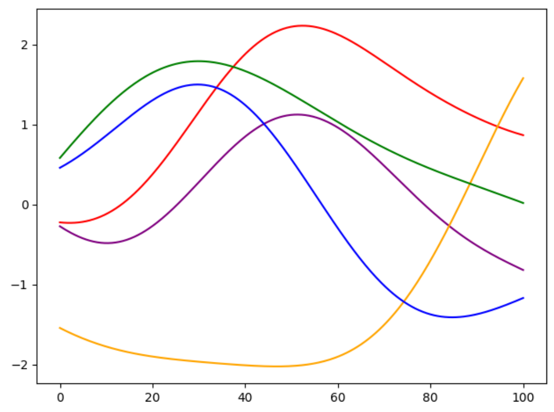
For a GP prior, $\mu(x) = 0$ is a typical assumption unless there is some specific reason to choose a different $\mu(x)$. There are a number of possible choices for k depending on the characteristics of the functions we wish to model. One common choice is the squared exponential kernel

$$k(x_1, x_2) = \sigma^2 \exp(\frac{-(x_1 - x_2)^2}{2l^2})$$

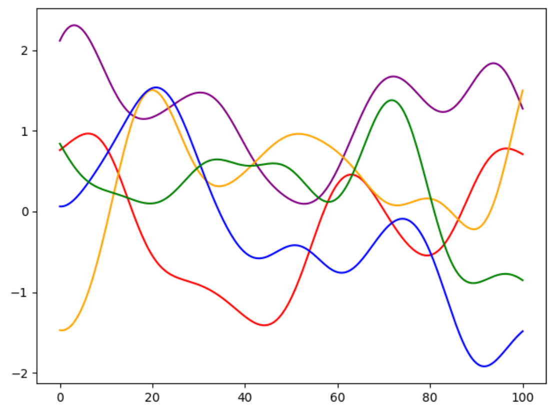
The functions it models are determined by the amplitude σ^2 and the lengthscale parameter l. σ^2 is a scale factor. l specifies how quickly the covariance between points x_1 and x_2 decays. Large values of l give rise to “slowly moving” functions while low values give rise to “quickly moving” functions.



Low lengthscale



High lengthscale

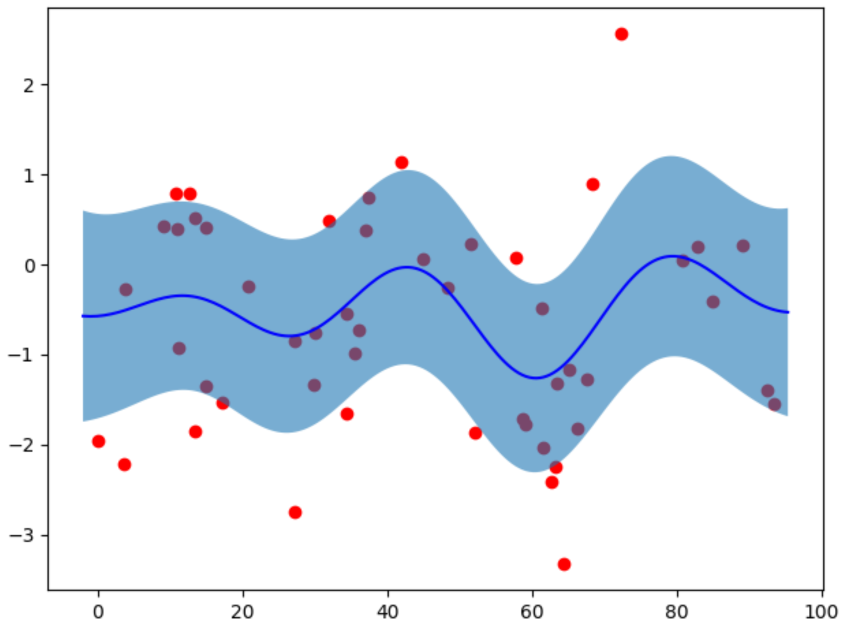


Medium lengthscale

Example draws from a Gaussian Process prior

This covariance function is versatile and can model many smooth functions. Other covariance functions give rise to functions with different behaviours. [5] For example, one can make use of a periodic kernel to model sinusoidal functions.

GPs are used in practice in a variety of contexts. A brief review of the literature on GPs finds them used for problems as far apart as reinforcement learning, object recognition and anomaly detection. For this reason, even a small improvement in GP training times or performance could have far-reaching effects. In the following sections, we will briefly touch on two broad uses of Gaussian Processes, specifically the cases of GP regression and GP classification with non-Gaussian likelihoods.



Example of GP regression in 2 dimensions. $\mu \pm \sigma$ plotted here.

2.2 Gaussian Process Regression

As seen above, a GP prior with an appropriate kernel encompasses a large class of non-linear functions. For this reason, GPs are often used for regression. Assuming that the underlying function can be modelled as a GP, we can assume that the training data and test data follow a joint Gaussian distribution. We also generally assume that our training data are not direct observations of the underlying function, but are contaminated by some Gaussian noise with mean 0 and variance σ_n^2. We can therefore write for training data $(x_1, f_1), ..., (x_n, f_n)$ and test point x_* the following.

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \\ f_* \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \begin{bmatrix} K(X,X)+\sigma_n^2 I & K(X,x_*) \\ K(x_*,X) & K(x_*,x_*) \end{bmatrix}) \tag{2}$$

Where $K(\cdot,\cdot)$ is the matrix or vector constructed by applying the kernel function repeatedly to pairs of points. As we know the training data, we can condition on it to obtain the marginal predictive distribution on our test point.

$$f_* \sim \mathcal{N}(\mu, \sigma^2)$$
$$\mu = K(X,x_*)^T (K(X,X)+\sigma_n^2 I)^{-1}\mathbf{f}$$
$$\sigma^2 = K(x_*,x_*) - K(X,x_*)^T (K(X,X)+\sigma_n^2)^{-1} K(x_*,x_*)$$

The fact that we can write the parameters of the predictive distribution in a closed form makes GP regression fairly easy to implement.

2.3 Non-Gaussian Likelihoods

For GP regression, the Gaussian form of the likelihoods of the outputs make inference easy. When using GPs to model outputs which do not have Gaussian likelihoods, we run into problems of tractability. We can take as an example the case of binary classification, but the problems we run into will occur for any use of GPs with non-Gaussian likelihoods. In binary classification, we model a latent function f with a GP prior and set the likelihoods $p(y_i|f_i) = \Phi(y_i f_i)$. We further assume the outputs y are either 0 or 1. If we write down the posterior distribution of the latent function f and the predictive distribution of the latent function f_* and the output y at a test point x_*, we end up with the following

$$p(\mathbf{f}|X,\mathbf{y}) \propto p(\mathbf{f}) \prod_{i=1}^{n} \Phi(y_i f_i)$$
$$p(f_*|D,\mathbf{x}_*) = \int p(f_*|\mathbf{x}_*,\mathbf{f}) p(\mathbf{f}|X,\mathbf{y}) d\mathbf{f}$$
$$p(y=1|D,\mathbf{x}_*) = \int p(f_*|D,\mathbf{x}_*) \Phi(f_*) df_*$$

The integral in the second equation becomes intractable due to the non-Gaussian likelihood terms in the first equation. We can circumvent this by approximating the likelihood terms with Gaussians $t_i(f_i) = \mathcal{N}(f_i; \mu_i, \sigma_i^2)$ whose parameters are learned by one of a class of algorithms. We will then have a Gaussian approximation to $p(\mathbf{f}|X,\mathbf{y})$ which will make the problem tractable.

We will make use of Expectation Propagation to learn these parameters later in the project and will expound further on our use of this algorithm in the appropriate section. One can also obtain these approximate parameters using Laplace Approximation, variational methods and other algorithms.

2.4 Active Learning

Active learning is a subset of machine learning in which the learning algorithm being used is allowed to choose which points it wants to query from the data-generating process. In other words, the algorithm is not fed training data "at random", but can choose to see the output at a particular point x in the input space. In this way, the algorithm can choose points that it is maximally uncertain about (under some metric) and thereby increase training speed. Active learning has been empirically shown to reduce training time and increase performance in many contexts. [9]

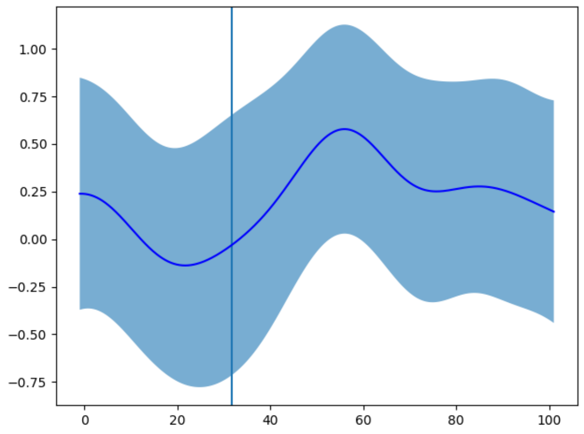


Figure 3: Simple example of active learning: the point with the highest posterior predictive variance is queried

There are a number of Active Learning algorithms for Gaussian Process models. One well-known example is Bayesian Active Learning by Disagreement (BALD)[6]. This algorithm queries the data-generating process at the point x which maximises the following objective function:

$$H[y|x, D] - \mathbb{E}_{\mathbf{f}\sim p(\mathbf{f}|D)}[H[y|x, \mathbf{f}]]$$

The original paper provides a nice intuition for understanding this objective function. We seek an x for which the entropy on the output y is high (in other words, an x for which we are highly uncertain about the corresponding output). We also, however, wish to pick an x which has a low expected entropy over values of the latent function $\mathbf{f}$. This can be thought of as picking a point x at which individual settings of $\mathbf{f}$ are highly certain but for which we are overall uncertain about the output. In other words, we pick a point at which we have a number of “hypotheses” $\mathbf{f}$ which explain the data well but which we do not have sufficient evidence to distinguish between. This is, at a high-level, the reason why it is most useful to have new evidence at this point. (temporary note: Is this interpretation accurate? I’m not 100% sure)

GPs can be very expensive to use, particularly for high-dimensional input spaces. Exact inference has a time complexity of $O(n^3)$ in the size of the training data. If using gridding for analysis, we have $O(G^D)$ points to consider, where G is the grid size and D the number of dimensions. Any increase in training efficiency is to be welcomed. This is one reason why active learning is particularly important for GP models.

3 Expectation Propagation

In this section we will discuss in detail the methods implemented for fitting GPs on mixed data.

3.1 Overview

Expectation Propagation is an iterative method for approximating probability distributions. We use it here to approximate the posterior distribution for a latent function $\mathbf{f}$ with a GP prior and training data with non-Gaussian likelihoods. We give first a brief general overview of EP before discussing our adaptation of the method for use with different types of data. Our overview here is adapted from GPML chapter 3 [1]. We suppress some of the details (the exact form of the parameters of q_{-i} etc.) and give these in the Appendix.

We assume that we have some data (x_i, y_i) generated from some latent function f which we wish to model using a GP prior. Further, we assume that the likelihoods $p(y_i|f_i)$ of these data under the latent function f are not Gaussian. As mentioned in section 2.3, the posterior predictive distribution of $\mathbf{f}(x_*)$ at a test point x_* is intractable. For this reason, we approximate the intractable likelihoods with “local likelihoods” $t_i(f_i)$ which are Gaussian. Note that the local likelihoods are functions of the latent function values f_i as we will be using them to approximate the posterior on $\mathbf{f}$.

We begin by initialising the parameters of our local Gaussian likelihoods $t_i(f_i \,|\, \tilde{\mu}_i, \tilde{\sigma}_i^2)$ with some reasonable values. Usually in implementation we deal with slightly different parameters $\tilde{\tau}_i = \tilde{\sigma}_i^{-2}$ and $\tilde{\nu}_i = \tilde{\tau}_i \tilde{\mu}_i$. These are usually initialised as $\tilde{\tau} = \tilde{\nu} = 0$. For this overview, we will talk about the parameters $\tilde{\mu}_i, \tilde{\sigma}_i^2$ as the transformation of these is just a detail of implementation. We will update these parameters iteratively, so at each step we will have an approximate Gaussian posterior on $\mathbf{f}$, $q(\mathbf{f} \,|\, X, y) = \frac{1}{Z} p(\mathbf{f} \,|\, X) \prod_{i=1}^{n} t_i(f_i \,|\, \tilde{\mu}_i, \tilde{\sigma}_i^2) = \mathcal{N}(\mathbf{f} \,|\, \boldsymbol{\mu}, \Sigma)$ where $p(\mathbf{f} \,|\, X)$ is the GP prior and Z is a normalising constant.

We update the parameters by looking at each t_i in turn. We first consider the marginal "cavity distribution", which is simply the marginal of f_i under our approximate posterior q with t_i "left out". In other words, we have the marginal $q_i(f_i \mid \mu_i, \sigma_i^2 = \Sigma_{ii})$ and from this we obtain the "cavity distribution" $q_{-i}(f_i \mid \mu_{-i}, \sigma_{-i}^2) = \frac{q_i(f_i)}{t_i(f_i)}$. The parameters of q_{-i} can be easily calculated as it is the ratio of two Gaussian pdfs.

We now consider a new (unnormalised) Gaussian marginal on f_i $\hat{q}(f_i \mid \hat{\mu}_i, \hat{\sigma}_i^2)$. We wish to choose the parameters $\hat{\mu}_i$, $\hat{\sigma}_i^2$ so that $\hat{q}$ best approximates the product of the cavity distribution q_{-i} and the exact likelihood $p(y_i \mid f_i)$. We do this by minimising $\mathrm{KL}(\hat{q}(f_i), q_{-i}(f_i)p(y_i \mid f_i))$. KL here means the Kullback-Leibler Divergence. KL is a measure of how much a given probability distribution differs from another. Minimising it here is equivalent to choosing the parameters of the new marginal distribution that best approximate the product of the cavity distributon and the exact likelihood. The KL divergence between two Gaussians is minimised when the first and second moments of these Gaussians match, so we pick $\hat{\mu}_i$ and $\hat{\sigma}_i^2$ to match the moments of the product $q_{-i}(f_i)p(y_i \mid f_i)$. Additionally, we match the zeroth moments to ensure that $\hat{q}$ is normalised. It is important to note here that this is the point at which EP diverges for different data types, as this is where we incorporate the exact likelihood $p(y_i \mid f_i)$

Once we have these new marginal moments, all that is left is to find the parameters of the likelihood approximation t_i such that the marginal of our full posterior q has these moments. This, again, can be fairly easily done due to the Gaussian forms of all the distributions involved. The exact form of the updated parameters is in the Appendix. [note have not written appendix yet!]

This completes the algorithm for a single step of EP. We pass over all the training data several times, updating iteratively, until we reach some predefined condition of convergence (for example, that the average distance between the new posterior mean and the old posterior mean falls below some ϵ).

Once we have reached convergence, we will have our posterior parameters $\boldsymbol{\mu}$, Σ on $\mathbf{f}$ for the training data and we have the following closed form formulae for performing inference at a test point x_*.

$$\mathbb{E}[f_*] = k_*^T K^{-1} \boldsymbol{\mu}$$

$$\mathbb{V}[f_*] = k(x_*, x_*) - k_*^T \Sigma^{-1} k_*$$

Where k_* is the vector generated by the kernel function applied to $x*$ and each of the training x in turn. All other quantities in the above formulae are as set out in the Background section.

3.2 EP for interval data

We now detail how we have adapted the above general algorithm for the case of interval data. First we clarify what we mean by interval data. Our training data has the form $(x_i, [a, b])$, where a and b denote the start and end points of the interval at x_i. The likelihood function here has the form $p(y_i \mid f_i) = p([a, b] \mid f_i) = \Phi(b; f_i, \sigma_n^2) - \Phi(a; f_i, \sigma_n^2)$ where, remember, f_i denotes the value of our latent function $\mathbf{f}$ at x_i and σ_n^2 denotes the Gaussian noise contaminating the data generated from the latent function.

In order to find our approximate posterior, we follow the broad outline of the EP algorithm as laid out in the previous section. As noted above, the point at which the outline above diverges for different data with different likelihoods is in the derivation of the marginal moments of $\hat{q}$. Until (and after) that point, the algorithm is essentially the same for any type of data. So here, we derive these moments for our interval likelihood.

Remember that we are looking for the zeroth, first and second moments of $q_{-i}(f_i)p(y_i \mid f_i)$ as we wish to to match these to $\hat{q}$. Let us define a new function $k(x)$ as follows:

$$k(x) = Z^{-1}\mathcal{N}(x; \mu_{-i}, \sigma_{-i}^2)(\Phi(\frac{x-a}{\sigma_n}) - \Phi(\frac{x-b}{\sigma_n}))$$

Where for the likelihood portion here we use the fact that $\Phi(b; x, \sigma_n^2) - \Phi(a; x, \sigma_n^2) = \Phi(\frac{b-x}{\sigma_n}) - \Phi(\frac{a-x}{\sigma_n}) = \Phi(\frac{x-a}{\sigma_n}) - \Phi(\frac{x-b}{\sigma_n})$. We do this to make the next parts of the derivation easier.

The moments of $k(x)$ here are precisely the moments we are looking for. We begin by finding the normalising constant Z.

$$Z = \int_{-\infty}^{\infty} \mathcal{N}(x; \mu_{-i}, \sigma_{-i}^2)(\Phi(\frac{x-a}{\sigma_n}) - \Phi(\frac{x-b}{\sigma_n}))dx$$

$$\implies Z = \int \mathcal{N}(x; \mu_{-i}, \sigma_{-i}^2)\Phi(\frac{x-a}{\sigma_n})dx - \int \mathcal{N}(x; \mu_{-i}, \sigma_{-i}^2)\Phi(\frac{x-b}{\sigma_n})dx$$

$$\implies Z = \Phi(z_a) - \Phi(z_b)$$

where $z_a = \frac{\mu_{-i}-a}{\sigma_n\sqrt{1+\frac{\sigma_{-i}^2}{\sigma_n^2}}}$ and $z_b = \frac{\mu_{-i}-b}{\sigma_n\sqrt{1+\frac{\sigma_{-i}^2}{\sigma_n^2}}}$

We now move on to finding the first moment. We do this by considering the derivative of Z with respect to μ_{-i}

$$\frac{\partial Z}{\partial \mu_{-i}} = \int \frac{x-\mu_{-i}}{\sigma_{-i}^2}\mathcal{N}(x; \mu_{-i}, \sigma_{-i}^2)(\Phi(\frac{x-a}{\sigma_n}) - \Phi(\frac{x-b}{\sigma_n}))dx = \frac{\partial}{\partial \mu_{-i}}(\Phi(z_a) - \Phi(z_b))$$

$$\implies \frac{Z}{\sigma_{-i}^2}\int xk(x)dx - \frac{Z\mu_{-i}}{\sigma_{-i}^2}\int k(x)dx = \frac{\mathcal{N}(z_a)-\mathcal{N}(z_b)}{\sigma_n\sqrt{1+\frac{\sigma_{-i}^2}{\sigma_n^2}}}$$

$$\implies Z\mathbb{E}_k[x] - Z\mu_{-i} = \sigma_{-i}^2\frac{\mathcal{N}(z_a)-\mathcal{N}(z_b)}{\sigma_n\sqrt{1+\frac{\sigma_{-i}^2}{\sigma_n^2}}}$$

$$\implies \mathbb{E}_k[x] = \mu_{-i} + \sigma_{-i}^2\frac{\mathcal{N}(z_a)-\mathcal{N}(z_b)}{Z\sigma_n\sqrt{1+\frac{\sigma_{-i}^2}{\sigma_n^2}}}$$

And finally we move on to finding the second moment. We do this by considering the second derivative of Z with respect to μ_{-i}.

$$\frac{\partial^2 Z}{\partial\mu_{-i}^2} = \int (\frac{x^2}{\sigma_{-i}^4} - \frac{2\mu_{-i}x}{\sigma_{-i}^4} + \frac{\mu_{-i}^2}{\sigma_{-i}^4} - \frac{1}{\sigma_{-i}^2})\mathcal{N}(x;\mu_{-i},\sigma_{-i}^2)(\Phi(\frac{x-a}{\sigma_n}) - \Phi(\frac{x-b}{\sigma_n}))dx = \frac{z_b\mathcal{N}(z_b)-z_a\mathcal{N}(z_a)}{\sigma_n^2+\sigma_{-i}^2}$$

$$\implies \int (x^2 - 2\mu_{-i}x + \mu_{-i}^2 - \sigma_{-i}^2)Zk(x)dx = \sigma_{-i}^4\frac{z_b\mathcal{N}(z_b)-z_a\mathcal{N}(z_a)}{\sigma_n^2+\sigma_{-i}^2}$$

$$\implies \mathbb{E}_k[x^2] - 2\mu_{-i}\mathbb{E}_k[x] + \mu_{-i}^2 - \sigma_{-i}^2 = Z^{-1}\sigma_{-i}^4\frac{z_b\mathcal{N}(z_b)-z_a\mathcal{N}(z_a)}{\sigma_n^2+\sigma_{-i}^2}$$

$$\implies \mathbb{E}_k[x^2] = 2\mu_{-i}\mathbb{E}_k[x] - \mu_{-i}^2 + \sigma_{-i}^2 + Z^{-1}\sigma_{-i}^4\frac{z_b\mathcal{N}(z_b)-z_a\mathcal{N}(z_a)}{\sigma_n^2+\sigma_{-i}^2}$$

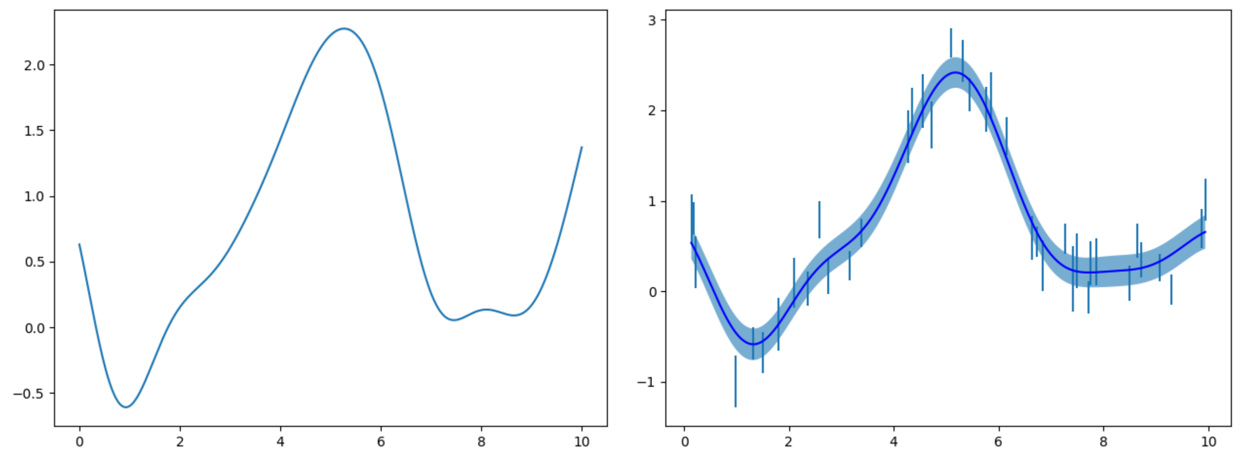
Now we find the centered moment.

$$\mathbb{E}_k[x^2] - \mathbb{E}_k[x]^2 = \sigma_{-i}^2 + \frac{\sigma_{-i}^4}{Z(\sigma_n^2+\sigma_{-i}^2)}(z_b\mathcal{N}(z_b) - z_a\mathcal{N}(z_a) - \frac{(\mathcal{N}(z_a)-\mathcal{N}(z_b))^2}{Z})$$

So we have found all of our moments and we can continue with EP as outlined above. We give an example visualisation of fitting to interval data below.

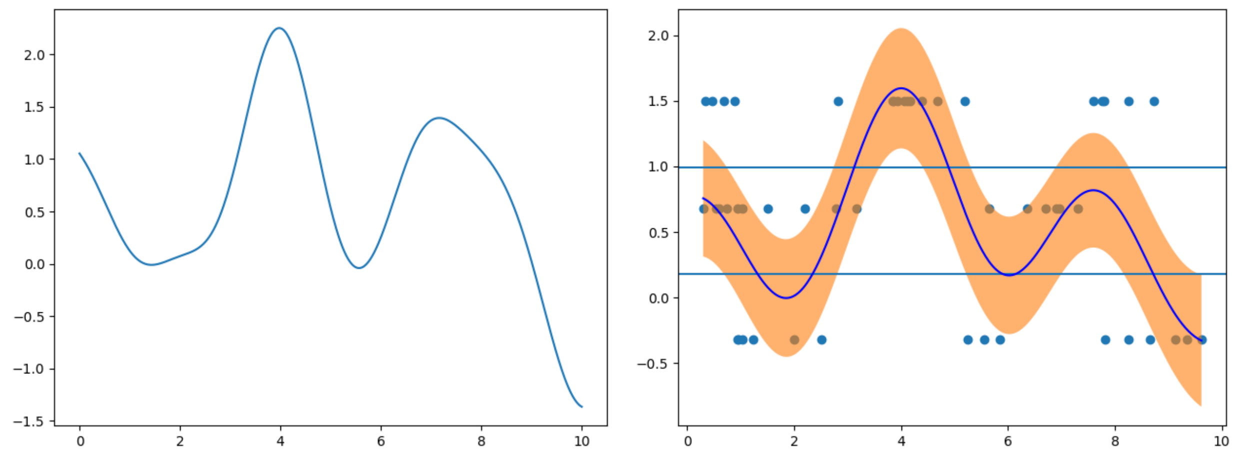
3.3 EP for ordinal data

We now detail how we adapted EP for use with ordinal data. Ordinal data here takes the form (x_i, y_i) where we have m ranks such that $y \in \{1, 2, \ldots, m\}$. We assume that the ranks are generated from some latent function f and some thresholds $\{t_0, t_1, \ldots, t_{m-1}, t_m\}$ such that $y_i = j$ if $t_{j-1} < f(x_i) \le t_j$. We also continue to assume that our latent function f is contaminated by some Gaussian noise with variance σ_n^2. Therefore our likelihood for these data is $p(y_i \mid f_i) = p([t_{y_i-1}, t_{y_i}] \mid f_i) = \Phi(t_{y_i}; f_i, \sigma_n^2) - \Phi(t_{y_i-1}; f_i, \sigma_n^2)$. So our likelihood is simply an interval likelihood as in section 3.2, and we can use the same moments as used in that section.


Original Data Fitted mean (with one $\pm\sigma$ in blue)

Example of EP for interval data

Theoretically we have that $t_0 = -\infty$ and $t_m = \infty$. In practice, this leads to numerical problems, so in our implementation we take t_0 and t_m to be some suitably large negative/-positive numbers on the scale of the amplitude. In our implementation, for an amplitude of 1, we take $t_0 = -10000$ and $t_m = 10000$. We also implemented a slightly different but equivalent formulation of GP regression for ordinal data [2] and found very similar results to our own implementation.


Original Data Fitted mean (with one $\pm\sigma$ in orange)

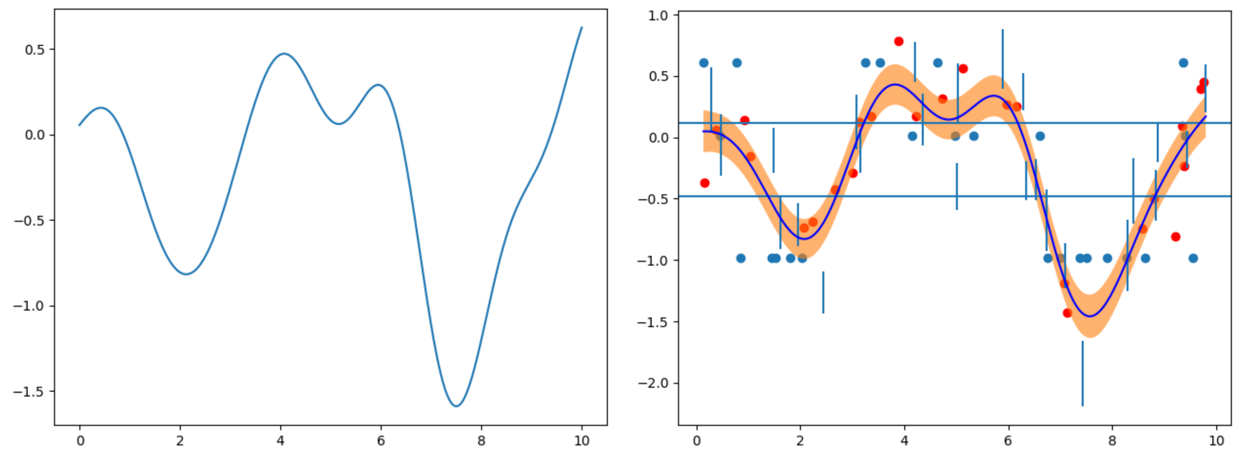
Example of EP for ordinal data (thresholds in blue)

3.4 Combining data

We now turn to the task of approximating a posterior in the case that our data (x_i, y_i) is a mixture of ordinal, interval and point data.

Note here by "point data" we mean points generated directly from the latent function $\mathbf{f}$ with the Gaussian noise with variance σ_n^2 applied. The likelihood for the point data is simply $p(y_i \mid f_i) = \mathcal{N}(f_i; y_i, \sigma_n^2)$. So the likelihood is Gaussian, and when our implementation of EP sees point data it simply directly sets $\boldsymbol{\mu_i} = y_i$ and $\Sigma_{i,:} = \Sigma_{:,i} = K(X, x_i)$ and additionally $\Sigma_{i,i} := \Sigma_{i,i} + \sigma_n^2$.

For the ordinal and interval training data, we simply perform EP as described above until convergence. Empirically, we see good results. A 2-d visualisation is below, but the algorithm works equally well for higher dimensional data.


Original Data Fitted mean (with one $\pm\sigma$ in orange)

Example of EP for mixed data (point data in red, ordinal in blue)

4 Artificial Data

In order to test our implementation of EP and our active learning algorithms, we must generate some suitable data. We do this by implementing a function to sample directly from the GP prior for a given length-scale and noise. We do this for a bounded input grid in a specified number of dimensions.

Once we have done this, we have a value of μ at each x_i in the input grid. Generating point data at a specified x_i is simply a matter of sampling from $\mathcal{N}(\mu_{x_i}, \sigma_n^2)$. Generating interval and ordinal data is a bit more involved.

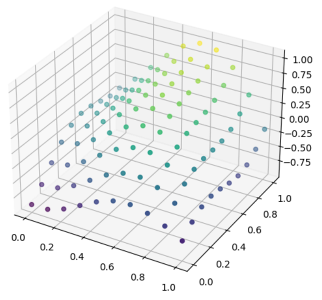
4.1 Ordinal Data

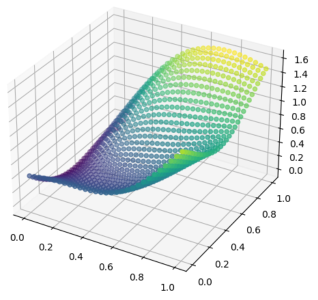
To generate ordinal data, we must first decide on a number of ranks r. Once we have done this, we calculate the r-quantiles of the set of generated μ and set our thresholds at these quantiles $q_1, q_2, \ldots, q_{r-1}$ with the highest and lowest threshold set as very high and very low values respectively (approximately "infinite" on the scale of the data). We can then sample an ordinal point at x_i by generating a point from $\mathcal{N}(\mu_{x_i}, \sigma_n^2)$ and assigning it a rank based on the thresholds we have calculated. If the generated point lies between threshold t_i and t_{i+1}, we assign it the rank $i + 1$.

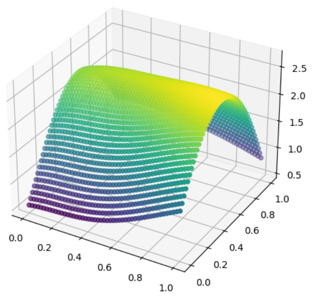
The choice of thresholds here is somewhat arbitrary. Any division of the output space could be chosen here. Randomly chosen thresholds were explored but tended to occasionally lead to degenerate behaviour (for example if thresholds were very close together), so for the sake of stable behaviour r-quantiles were decided upon. Different types of thresholds could be an interesting avenue for further research in this topic.

4.2 Interval Data

To generate interval data, we first fix an interval length l. We then choose M equally spaced points $a_1, \ldots, a_m$ in the output space between bounds $(-z, z - l)$. We now have M intervals with bounds $[a_i, a_i + l]$. We assign a likelihood $\Phi(\frac{a_i + l - \mu_{x_i}}{\sigma_n}) - \Phi(\frac{a_i - \mu_{x_i}}{\sigma_n})$ to each of these M intervals. We then normalise by dividing each likelihood by the sum of the likelihoods. For large enough values of M we now have an approximation to a distribution of intervals in the output space for the generated dataset.


Low grid size


Medium grid size


High grid size

Example of 2-D artificial generated data with varying grid size

5 Active Learning for Mixed Data

Now that we have a method for approximating a posterior for a GP model with mixed data, we can move to the problem of active learning for mixed data. We state the problem we treat in this project as follows.

5.1 Problem statement

We are trying to fit a function modelling a data generating process (DGP) which can be modelled with a Gaussian Process prior (i.e. the function is continuous in the input). We can request an output $y_d(x)$ at a point x from the DGP at each of timesteps $1, \ldots, T$ until some terminal time T. Further, we can specify the datatype d we wish to see at x. We have some total budget C. We can request ordinal data with cost c_o, interval data with cost c_i or point data with cost c_p. At each timestep t, how do we decide the datatype d and the input x for which to request an output?

Ordinal data will tend to provide less information than interval data, which in turn will tend to provide less information than point data. We wish to find a suitable algorithm which can balance cost against information gain at each timestep. In order to do this, we must find some objective function which measures a notion of "expected information gain" at a given point for a given data-type. We must also ensure that this function takes into account our budgetary constraints.

5.2 BALD

One possible candidate as an algorithm is the one laid out in the paper Bayesian Active Learning by Disagreement (BALD). This was briefly discussed in the background section. BALD empirically performs very well in a number of contexts [6] [8] and has served as a basis for adaptation to other problems [7]. It would appear to satisfy our requirement for an objective function that measures information gain. However, we run into the issue that BALD's objective function

$$H[y|x, D] - \mathbb{E}_{\mathbf{f}\sim p(\mathbf{f}|D)}[H[y|x, \mathbf{f}]]$$

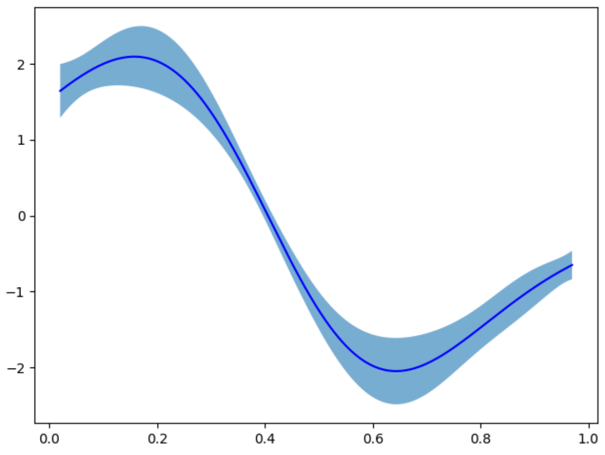
is defined in terms of a discrete output y. We cannot directly calculate this objective function for our point and interval data types. Instead, we must approximate it.

For point data, we divide the output space into M equal sized buckets with thresholds at $a_1, \dots, a_{M+1}$, with bounds at $(-b, b)$. We can choose the bounds to have some suitable values in terms of the amplitude, for example if the amplitude is 1 we can pick $b = 3$ and the buckets should nearly always contain all possible generated point data. We now divide the input space into a grid $x_1, x_2, \dots, x_n$. We generate N samples from our posterior on $\mathbf{f}$ on the grid. We now have N samples of μ at each $x_1, \dots, x_n$ from our posterior.

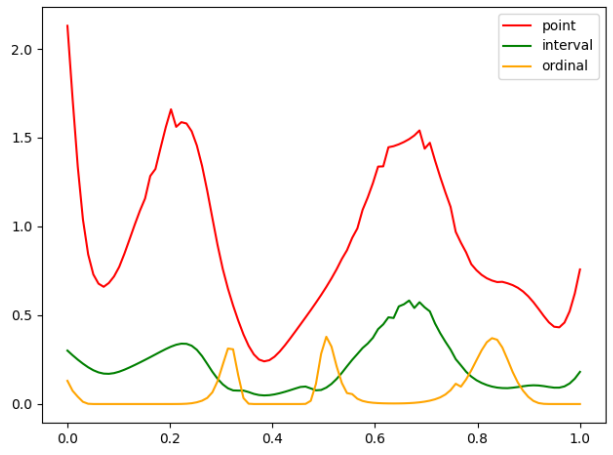
We can calculate $H[y|x_i, D]$ by calculating the average probability mass in each bucket for each value of μ at x_i and calculating the entropy on the M buckets. We can similarly calculate $\mathbb{E}_{\mathbf{f}\sim p(\mathbf{f}|D)}[H[y|x_i, \mathbf{f}]]$ by calculating the entropy on each of the M buckets for each value of μ first and then averaging.

We do something similar for interval data. First, we fix an interval size l. We then divide the output space with M equally spaced thresholds between $(-b, b-l)$. We take our "buckets" here to be the intervals $[t, t+l]$ where t is one of our M thresholds. We then once again take N samples from our posterior on a grid $x_1, \dots, x_n$. We calculate the likelihood of each bucket $[t, t+l]$ with the interval likelihood $\Phi(\frac{t+l-\mu_{x_i}}{\sigma_n}) - \Phi(\frac{t-\mu_{x_i}}{\sigma_n})$. We then normalise the likelihoods to obtain a probability distribution over our M discretised intervals. We can use this distribution to calculate the quantities for the BALD criterion as above.

Ordinal data is the easiest as it does not require discretisation. The "buckets" are naturally defined by the rank thresholds. We can proceed as above by taking N samples, directly calculating the probability mass for each bucket and calculating the two quantities necessary for the BALD criterion.



Fitted function



BALD criterion for each data-type

Example of vanilla BALD criterion

The visualisation of the BALD criterion above matches our intuition about an objective function which seeks to maximize information gain. The BALD criterion is highest at points where uncertainty is highest. It dips where uncertainty is low. It it generally higher for data-types that provide more information (point data in particular). The ordinal BALD spikes where the fitted function is near an ordinal threshold and is close to zero when the function is far from a threshold.

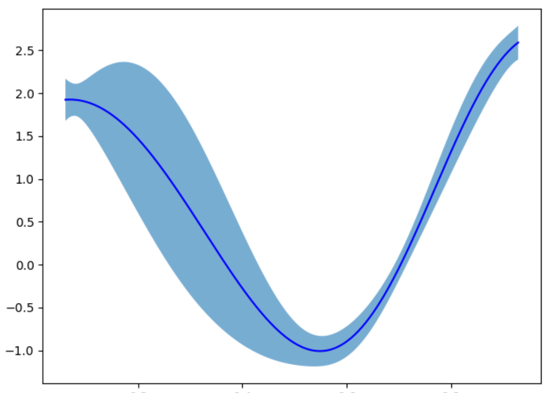
5.3 Adjusting for cost

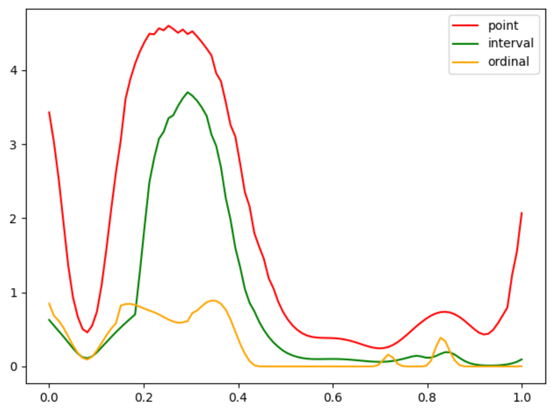
Directly using the BALD criterion does not work for our full problem statement. This is because certain data-types tend to carry more information than others. As can be seen in the graph above, point data taken directly from the underlying function is nearly always associated with a higher BALD criterion than the other two data-types. If we do not adjust for this fact, then our learning algorithm will simply pick point data at every time step. This would make sense if we did not have any budgetary constraints, but we are here assuming that we do. We must find a way to adjust for the cost of querying a certain data-type.

The most obvious route to achieving this is direct cost-adjustment by simply finding the ratio of BALD criterion to cost for each data-type and using this as our new criterion. In this case, we obtain a new quantity which can be interpreted as "information gain per unit cost". We will call this **cost adjusted BALD criterion.** At each time step, we now seek the point x and datatype d according to the following expression:

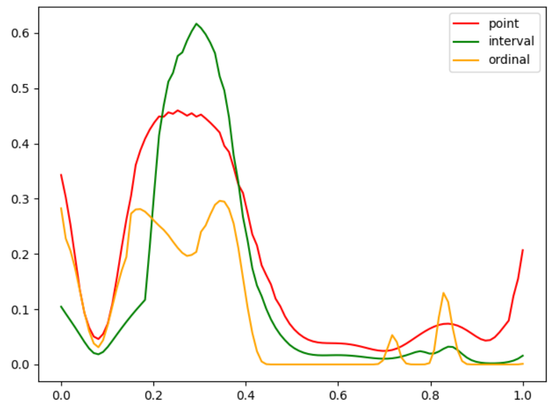
$$\arg\max_{x,\,d}\left(\frac{ordinalBALD(x)}{c_o}, \frac{intervalBALD(x)}{c_i}, \frac{pointBALD(x)}{c_p}\right)$$

Where the 3 BALD functions in the expression above are simply defined for their respective data-types as in section 5.2.


Fitted function



Vanilla BALD criterion



Cost-adjusted BALD criterion

Example of cost-adjustment (here $c_p = 10, c_o = 3, c_i = 6$)

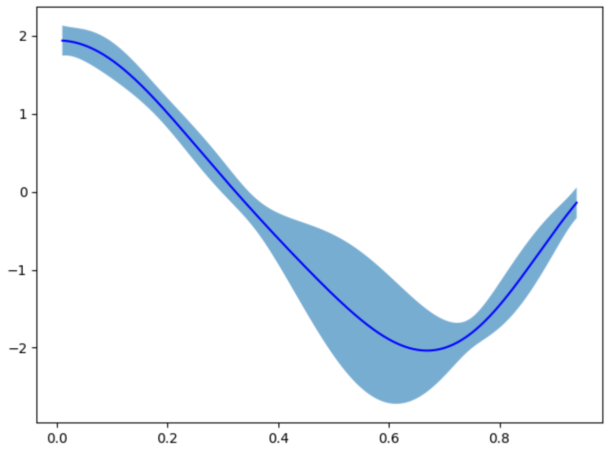
We can see that cost-adjustment leads to a situation where the objective function's landscape isn't completely dominated by the most informative data-type. It is important here to note that for particular settings of the costs, we will recover the behaviour of vanilla BALD (for example, if point data has a lower cost than the less informative data-types or if all data-types have the same cost). In visualisations and testing here we will tend to focus on the more "interesting" cases where cost-adjustment makes a meaningful difference to the algorithm's behaviour. This will be further expounded upon in the results section.

We could also come at the problem from a different angle. Our issue is the fact that the BALD criterion is "inherently" higher for certain data-types. We could account for this fact by taking the ratio of BALD criterion to the maximum theoretically possible BALD criterion. This corresponds to the entropy of the discrete distribution for a particular data-type being maximized, while the expected entropy of the discrete distribution over the posterior is minimised. So we can take the entropy as being that of the uniform distribution over the number of buckets, while taking the expected entropy under the posterior as 0. So we end up with a new quantity (where M is the number of buckets when performing discretisation as above):

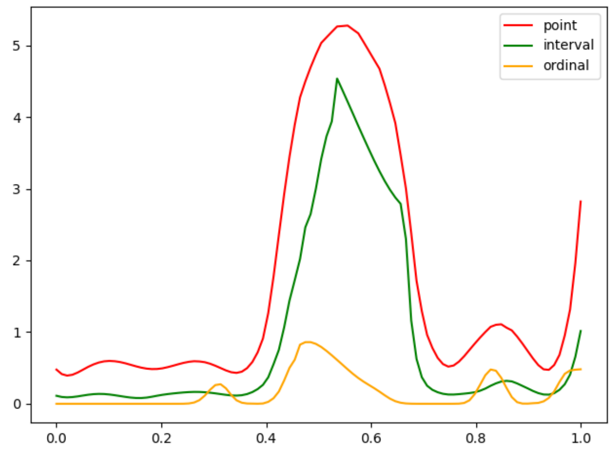
$$\frac{H[y|x, D] - \mathbb{E}_{\mathbf{f} \sim p(\mathbf{f}|D)}[H[y|x, \mathbf{f}]]}{\log_2(M)}$$

We will call this quantity the **normalised BALD criterion**. It can be interpreted as being the proportion of the maximum possible information gain from the specified data-type. This has the advantage of making the BALD criterion more obviously commensurable between data-types.

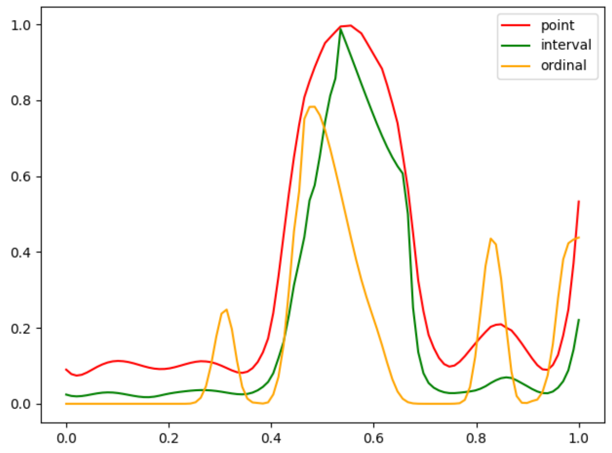
In the figure on page 22, we can see that normalisation similarly “shifts” the BALD criterion so that one data-type does not dominate the objective function’s landscape. We can think of the maximum of the normalised criterion over the three data-types as representing the point at which a particular data-type is “particularly informative” relative to what you would expect for that data-type. So maximizing the normalised criterion means that we are now looking for points at which e.g. ordinal data is “particularly” informative above what you would expect etc.



Fitted function



Vanilla BALD criterion



Normalised BALD criterion

Example of normalisation

6 Results

We must define a baseline algorithm to query data at each timestep against which to judge the two algorithms described above. We also must specify certain conditions under which to test.

6.1 Baseline

We specify two baselines here.

The first is a totally naive querying regime in which we pick a random point x and a random datatype to query at each stage. A comparison between our algorithms and this baseline will serve as a useful sanity check. If our algorithms do any worse than this baseline in any context then they do not add any value whatsoever.

The second does away with the datatype selection. It simply picks a datatype at random and then makes use of the BALD criterion for that datatype to pick an x to query at. Our expectation here is that this baseline performs significantly better than the totally naive querying regime but worse than the algorithms laid out in section 5. We will refer to this baseline as our “better baseline”.

6.2 Normalisation vs. Cost-Adjustment

Computing the BALD criterion for a large grid-size is quite time intensive. This is because we require N samples from M points, so the computational complexity increases on the order $O(MN)$. Further, M increases exponentially with the dimension of the input space. If we wish to take samples for a given artificial dataset with lengthscale l, a rough lower bound on the number of points required to sufficiently define the function is to take input points such that the furthest distance between two input points is l. This is because the lengthscale can be interpreted as being the minimum distance required for the function to change “significantly”. Taking this rule of thumb, we could say that a rough lower bound for the number of points we must consider, M, would be $O(\frac{G^D}{l})$ where G here is the length of the bounds we consider in each dimension (assume G is constant across dimensions here). So before doing more extensive testing, we should decide on which of the methods laid out in section 5 seems more likely to achieve interesting results.

It is not immediately obvious which of the two methods laid out in section 5 is most promising. We can test the two methods against one another by comparing their performances in a particular testing regime. We must first decide on a suitable set of conditions under which to test.

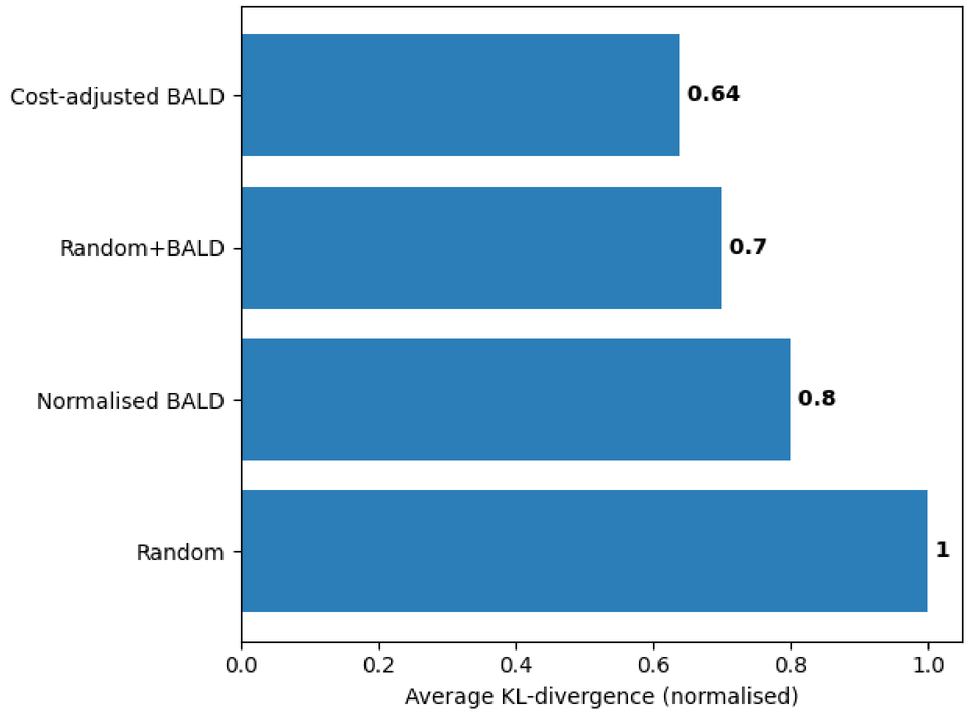
It was decided to perform a preliminary test with a two-dimensional input space with a lengthscale of 0.2 and bounds of (0,2) in each dimension. Further, in order to check that our algorithms give expected behaviour over a variety of costs, we performed 10 runs each for the 54 combinations of point, ordinal and interval costs where $c_p \in \{8, 9, 10, 11, 12\}, c_o \in \{2, 3, 4\}, c_i \in \{6, 7, 8, 9\}$ and point costs were higher or equal to interval costs. These cost ranges were decided upon to test the algorithms where behaviour was "interesting".

For lower values of point cost, the cost-adjustment algorithm always picks point data and its behaviour collapses to that of vanilla BALD. For higher values, the cost-adjustment algorithm always picks the cheaper data types. In practice, this behaviour makes sense (if the most informative data type is cheap, you should pick it). However we wish to study cases where the cost-adjustment algorithm's behaviour diverges from vanilla BALD. Several cost regimes were manually tested before the formal comparison runs and the costs in the formal comparison runs were set to roughly span the lower and upper bound of "interesting" behaviour from the cost-adjustment algorithm in this particular context.

The metric by which we compare our algorithms is KL-divergence from the ground truth data as a proportion of baseline. The reason we do this is that the baseline should always have the largest KL-divergence. It should in theory always perform worse than our active learning algorithms. Therefore it makes sense to compare the performance of our learning algorithms as a proportion of the "worst possible" performance. Below we lay out clearly the steps we take when performing a "run" for each of our algorithms.

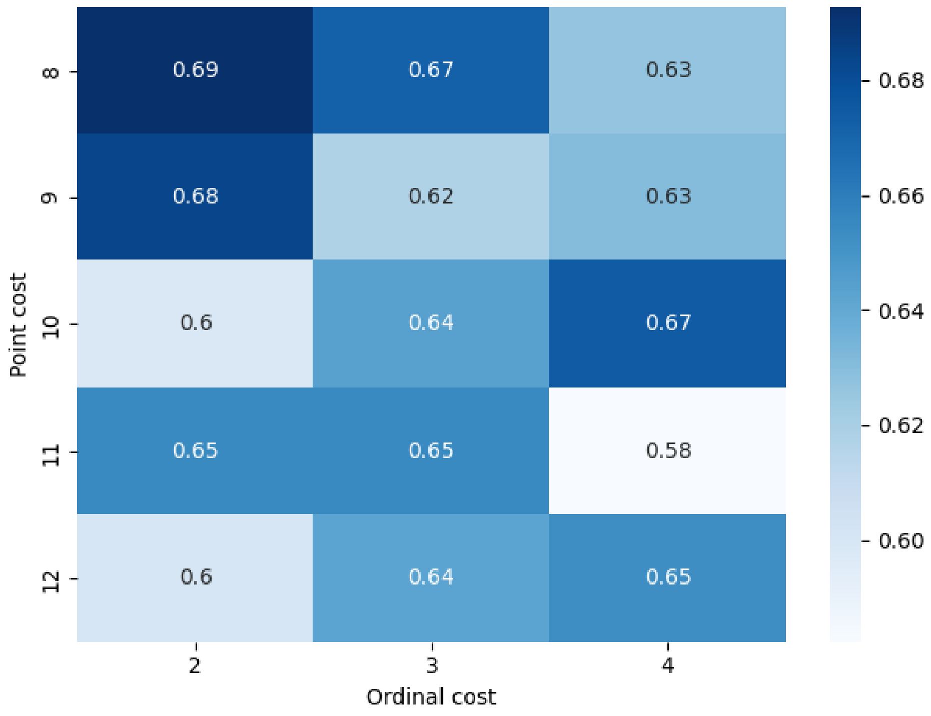
1. Generate some ground truth data from a Gaussian Process prior
2. Generate some initial training data from the ground truth (here we take 3 randomly selected points with noise)
3. Create an instance of a Gaussian Process object for each learning algorithm/baseline. This will store the training data sampled by each learning algorithm and perform EP.
4. Find the posterior distribution for each algorithm for this initial data using EP
5. Sample from the ground truth data according to each of the active learning algorithms we are testing
6. Find the new posterior distributions with this new data
7. Continue 4 & 5 until every GP object's budget is exhausted
8. Calculate the KL-divergence between the ground truth data and our final posterior distributions.
9. Normalise the KL-divergence so that the Random baseline has a divergence of 1.

Each algorithm had a budget of 60. For each run, termination occurred when all algorithms exhausted their budgets or at timestep 20, whichever came first. The table below shows the average results over all runs for each algorithm aggregated over all cost settings.



We should interpret these results with caution. They show performance in a particular context preset by the author. In this context, we see that cost-adjusted BALD performs the best out of our 540 runs. Reassuringly, it performs better than both baselines. Normalised BALD performed worse than randomly selecting a datatype and then selecting an input point with vanilla BALD, but significantly better than completely random selection of input points. Normalised BALD remains attractive as a path for further study. It provides a metric with a nice interpretation, and clearly performs better than a very simple baseline. There is likely a way to combine it cost-adjustment to improve its performance, but we will not explore that here.

Below we split our results for our 540 runs by ordinal and point cost. We average out the results over interval cost values. The numbers here are again expressed as a proportion of KL-divergence for the random case. There is no clear pattern in performance by cost, implying that our algorithm gives improved performance for a range of costs. This is what we would expect to see, and is reassuring.



We see a similar pattern when generating a heatmap for the other 3 combinations of costs, but omit these heatmaps for brevity here. A possible future avenue of research is to see how performance fares over a much larger range of costs.

We also tested our algorithms to see if dimensionality of input had an effect on their performance. Here we test our "better" baseline against cost-adjusted BALD. We did approximately 200 runs for artificial data in 2 and 3 dimensions. These runs were again distributed evenly

We found that cost-adjustment improved performance over our "better" baseline for both 2 and 3 dimensional data. We achieved similar results in both cases, with cost-adjusted BALD achieving an approximate reduction of **10%** in average KL-divergence over our baseline in both 2 and 3 dimensional input spaces. This is also consistent with the results we see above and implies our algorithm works for higher dimensional input spaces.

6.3 Example applications to real data

We conclude our analysis of the performance of our active learning algorithm by applying it to two real-world datasets. Both datasets are taken from the UCI repository [4]. We wish to check if the advantages we see in artificial data is borne out in real data. This also gives us an opportunity to check how our algorithm performs with regard to sample efficiency. A sample efficient algorithm requires fewer data points to reach a good level of performance.

If our algorithm improves sample efficiency, we should expect to see it converge on "good performance" with less data than baseline.

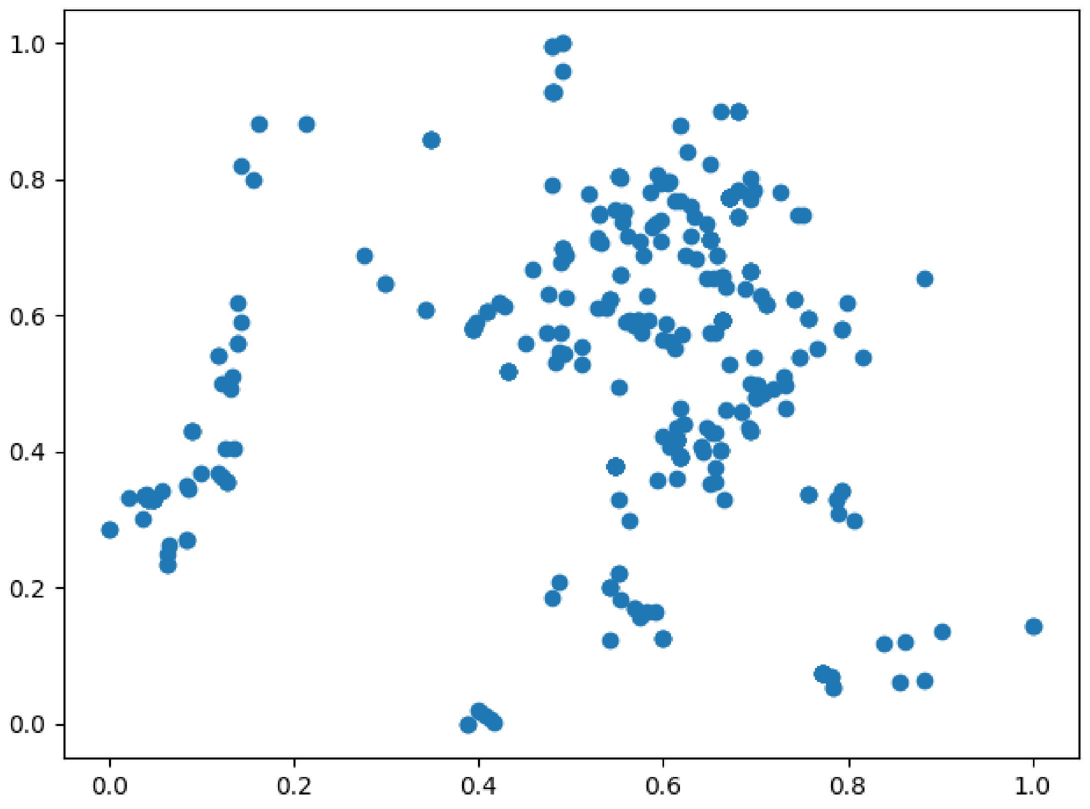
Gaussian Process models lend themselves particularly well to cases where the input space has some natural notion of distance. For this reason, both of the datasets we analyse here will have two dimensional input spaces corresponding to latitude and longitude in both cases. We note here that it is quite difficult to directly apply the methods we've derived above on observational datasets available online. This is because one of the assumptions of our algorithm is that we can freely request data at **any** input point from the DGP. When looking at passive observational datasets where we do not have the ability to directly interact with the DGP, this is not possible. We work around this constraint below, but it is something to keep in mind.

We begin with a dataset of real estate prices from a city in Taiwan [10]. The full dataset has a number of inputs and an output of house price per unit area.

We begin by selecting two inputs to focus on, these being latitude and longitude as mentioned above. Preliminary study shows that our output varies significantly with these two inputs. We normalise our price and our two inputs. We now have three attributes which are bound between 0 and 1.

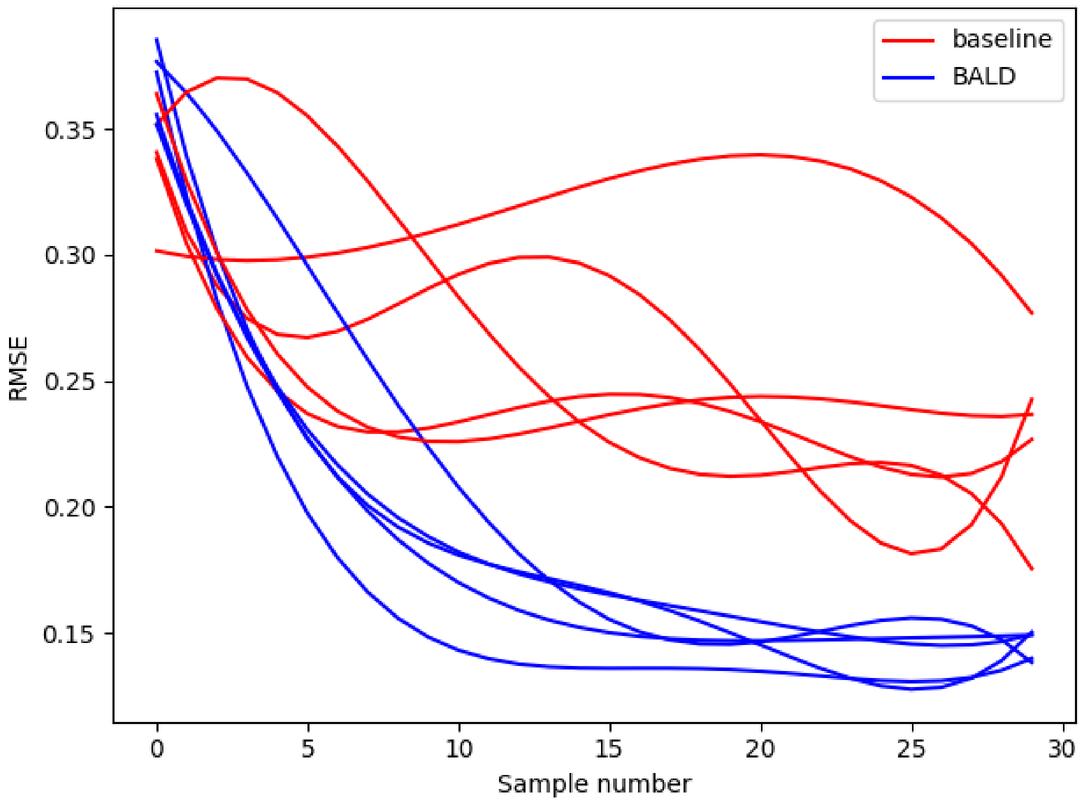
As before, we can model this as a GP Regression problem. Our output is, however, simply a real number between 0 and 1. We do not have mixed outputs, and therefore have no obvious way to apply the methods derived above. We can fix this fairly easily by assuming that the outputs are our "ground truth" as above and modifying the outputs to give an interval or ordinal output when requested.

Our inputs (normalised latitude and longitude) are distributed as follows:



Instead of calculating BALD for each point on a grid as before, we only calculate BALD at the available input points. This is because we don't have the option of seeing an output at any other input point as we do not have access to any sort of full posterior distribution in the input space. We fix the cost for each datatype at a particular setting ($c_p = 10, c_i = 7, c_o = 3$) and perform 5 runs of the random baseline and the cost-adjusted BALD algorithm. Each algorithm has a budget of 200 and can sample a maximum of 30 datapoints.

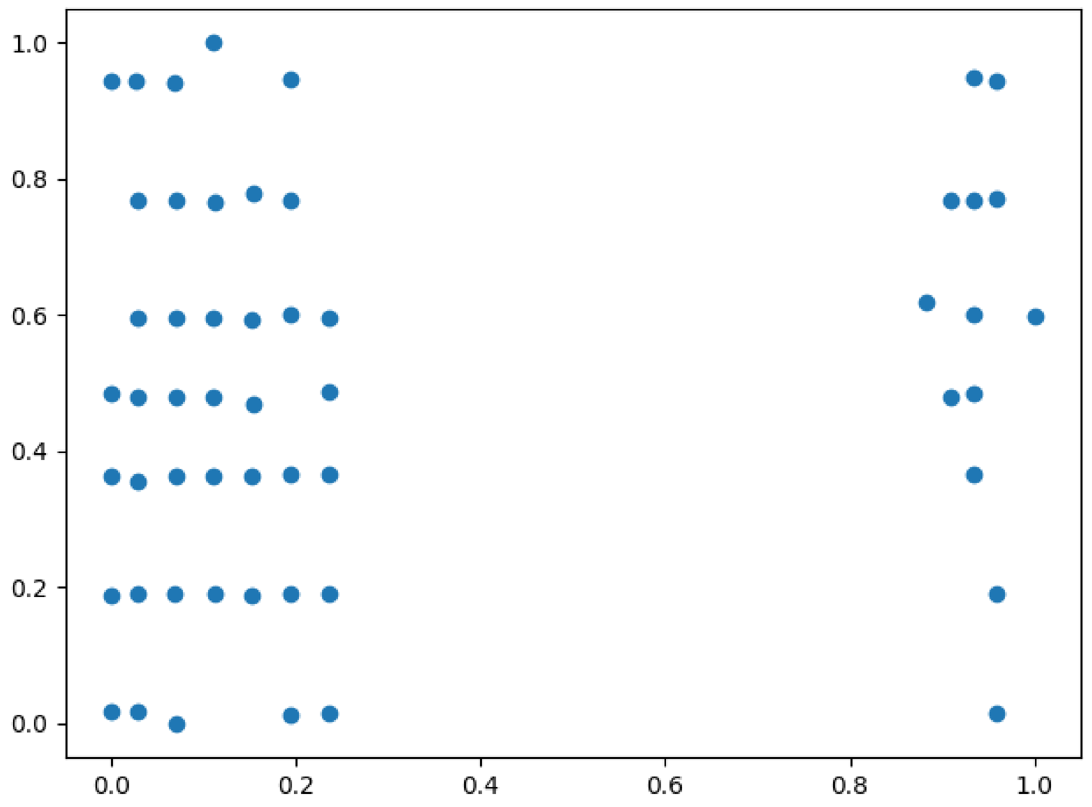
We have seen above that performance is not particularly sensitive to cost settings, so the cost setting we used is simply the average of those used in the previous section. We set aside 20% of the data for use as a test set. Our performance metric here is Root Mean Squared Error on the test set.



Above is an interpolation of the results for each of the 5 runs for baseline and the cost-adjusted BALD algorithm. Baseline is in red while BALD is in blue. We can see that BALD converges to an RMSE of about 0.15 which is significantly better than RMSE of about 0.25 that is achieved by baseline. Further, BALD achieves its better performance more quickly than baseline. In other words, we can see that our algorithm provides improved sample efficiency versus a simple baseline.

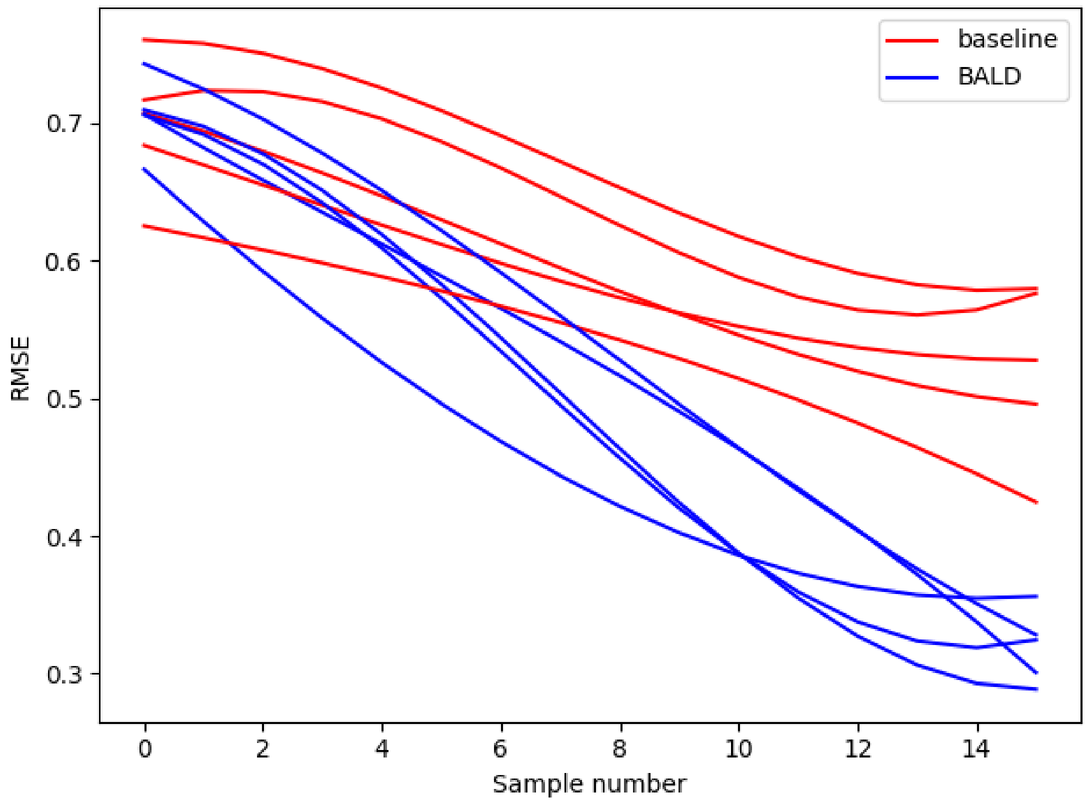
We now move on to our second dataset. This consists of air temperature readings at approx. 60 buoys in the Pacific Ocean, along with the coordinates of each buoy[3]. Temperature is our output here, and latitude and longitude our inputs. We perform normalisation as above.

Distribution of input points (buoys) with normalised latitude/longitudes:



The positions of the buoys above are approximately on the coast of several islands in the Pacific, which also explains the large gap in input coverage. Our method should still work as we only calculate BALD for those input points we know we can query.

We maintain the same cost settings and budget as above. We set the test set for each run to be 33% of the total dataset. We set a maximum number of points to query at 15 as our dataset is smaller in this case. We again use RMSE as our performance metric. We also once again perform 5 runs of baseline and our algorithm and give an interpolation of the results below.



We again find very similar results to the previous dataset analysed. Sample efficiency is improved when using our algorithm over baseline.

7 Conclusion

In this project, we have extended the standard method of Expectation Propagation to the task of Gaussian Process Regression in the case where training data is mixed. We proceeded to propose a method for active learning in the context of Gaussian Process Regression for mixed data and evaluate the performance of that method. There are a number of things we did not have time to investigate or test in the course of this project, and we detail some of these below.

We made the assumption that our hyperparameters were fixed when evaluating performance and doing EP. There exist methods for finding the hyperparameters for a GP model (for example, by using variational methods) and in principle it should be relatively straightforward to extend the work in this project to include those methods.

We also limited the scope of our work to three data types. In principle, extending the work here to other data types should be relatively easy. We calculate our BALD quantities here by simple sampling from the posterior, so once we have acquired an approximate posterior distribution conditioned on our training data, the active learning methods described here would be straightforward to adapt. The main work for taking account of new data types would involve modifying the moments for expectation propagation to take account of the new likelihood.

The original BALD paper [6] made use of some approximation tricks to speed up their calculations. Their approximations were for the example of GP binary classification and did not apply to the data types we studied here. It is likely, however, that some simple approximation tricks exist to speed up our implementation of BALD. Unfortunately we did not have time to find these and instead relied on direct sampling from the posterior, which was quite slow and hindered our ability to do larger scale testing of our algorithms.

In the real world, one may have to make decisions about which points/datatypes to query as a batch decision in advance. For example, extending the example from the introduction, a lab may have to make the decision to order 10 tests at 10 different locations at once, rather than being able to sequentially order tests after seeing the results of previous tests. There has been some work done on similar problems in the past [7], but we are not aware of any work done on this problem in the context of mixed data. This could be a fruitful avenue for further research.

References

[1] Rasmussen C.E & Williams C.K. *Gaussian Processes for Machine Learning.* MIT Press, 2006.

[2] Wei Chu, Zoubin Ghahramani, and Christopher KI Williams. "Gaussian processes for ordinal regression." In: *Journal of machine learning research* 6.7 (2005).

[3] Di Cook. *El Nino Data.* 1999. URL: `https://archive.ics.uci.edu/ml/datasets/El+Nino`.

[4] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository.* 2017. URL: `http://archive.ics.uci.edu/ml`.

[5] David Duvenaud. *The Kernel Cookbook.* URL: `https://www.cs.toronto.edu/~duvenaud/cookbook/`.

[6] Neil Houlsby et al. "Bayesian Active Learning for Classification and Preference Learning". In: (2011). arXiv: `1112.5745 [stat.ML]`.

[7] Andreas Kirsch, Joost Van Amersfoort, and Yarin Gal. "Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning". In: *Advances in neural information processing systems* 32 (2019).

[8] Gaurav Kumar and Venu Govindaraju. "Bayesian active learning for keyword spotting in handwritten documents". In: *2014 22nd International Conference on Pattern Recognition*. IEEE. 2014, pp. 2041–2046.

[9] Burr Settles. *Active learning literature survey.* University of Wisconsin-Madison Department of Computer Sciences, 2009.

[10] I-Cheng Yeh and Tzu-Kuang Hsu. "Building real estate valuation models with comparative approach through case-based reasoning". In: *Applied Soft Computing* 65 (2018), pp. 260–271.

8 Code

We present the code used in this project in sections. We also omit some of the most verbose visualisation code used for making plots etc. Python was the only programming language used in the course of this project.

8.1 Gaussian Process class

Python uses whitespace to delimit function/class code which presents a bit of a problem when attempting to fit code into a LaTeX file. The code below should be wrapped in a GaussianProcess class definition, but we suppress the explicit class declaration below to avoid the necessary whitespace and resulting difficulty in fitting the code onto an A4 page.

```
import numpy as np
from scipy.spatial import distance_matrix
from scipy.stats import norm
from numpy.linalg import cholesky, solve, inv
from matplotlib import pyplot as plt

def __init__(self, l=1, noise=1, dim=1, amplitude=1, b=[], budget=0):

    # set hyperparameters

    self.l = l
    self.noise = noise  # noise variance
    self.dim = dim  # dimension of data
    self.amplitude = amplitude

    self.mu = None
    self.tau = None

    self.train_data = {'interval':None, 'ordinal':None, 'point':None}
    self.all_x = None

    self.budget=budget


    self.b = [-1000000] + b + [1000000]


def add_train_data(self, data, data_type):

    assert isinstance(data, np.ndarray)
    assert data_type in ['point', 'ordinal', 'interval']
    assert (data_type in ['point', 'ordinal']) == (data.shape[1] == self.
                                               dim + 1)  # if data is point/
                                               ordinal, ensure right size
    assert (data_type == "interval") == (data.shape[1] == self.dim + 2)  #
                                                same for interval
    if self.train_data[data_type] is not None:

        self.train_data[data_type] = np.append(self.train_data[data_type],
                                               data, axis=0)

    else:

        self.train_data[data_type] = data

    validkeys = [k for k in ["interval", "point", "ordinal"] if self.
                                         train_data[k] is not None]

    self.all_x = [x[0:self.dim] for k in validkeys for x in self.
                                         train_data[k]]

def kernel(self, X1, X2):

    X1 = X1.reshape(-1, self.dim)
    X2 = X2.reshape(-1, self.dim)
    distances = np.square(distance_matrix(X1, X2)).astype("float")
    return self.amplitude * np.exp(-distances/(2*self.l**2))

def Full_EP(self, max_iter=10):

    all_data = []
    data_types = []

    if self.train_data["interval"] is not None:

        a = self.train_data["interval"][:, self.dim]
        b = self.train_data["interval"][:, self.dim+1]

        for l in self.train_data["interval"]:
            all_data.append(l)

        data_types = data_types + ["interval"]*(len(self.train_data["
                                              interval"]))

    if self.train_data["point"] is not None:

        for l in self.train_data["point"]:
            all_data.append(l)
        data_types = data_types + ["point"]*(len(self.train_data["point"])
                                              )

    if self.train_data["ordinal"] is not None:

        for l in self.train_data["ordinal"]:

            all_data.append(l)
        data_types = data_types + ["ordinal"] * (len(self.train_data["
                                               ordinal"]))


    X = np.array(self.all_x)
    n = len(X)

    tau = np.zeros([n,1])
    v = np.zeros([n,1])
    K = self.kernel(X, X)

    Sigma = K.reshape([n,n])

    mu = np.zeros([n,1])
    m = np.zeros([n,1])

    converged = False
    iteration_count = 0

    while not converged and iteration_count < max_iter:

        iteration_count += 1
        prev_mu = mu

        for i in range(n):

            if data_types[i] == "interval":

                tau_cav = Sigma[i, i]**-1 - tau[i]
                v_cav = mu[i]*(Sigma[i, i]**-1) - v[i]

                mu_cav = v_cav/tau_cav
                sig_cav = 1/tau_cav

                z_a = (mu_cav - a[i]) / ((self.noise ** (1 / 2)) * (1 + (
                                                    sig_cav / self.noise)
                                                    ) ** (1 / 2))
                z_b = (mu_cav - b[i]) / ((self.noise ** (1 / 2)) * (1 + (
                                                    sig_cav / self.noise)
                                                    ) ** (1 / 2))
                Z = norm.cdf(z_a) - norm.cdf(z_b)

                mu_hat = mu_cav + ((sig_cav/(Z*(self.noise**0.5)))*(1/(1+(
                                                    sig_cav/self.noise))
                                                    **0.5))*(norm.pdf(z_a
                                                    ) - norm.pdf(z_b))

                var_hat = sig_cav + (sig_cav**2)*(1/(Z*(self.noise+sig_cav
                                                    )))*(z_b*norm.pdf(z_b
                                                    ) - z_a*norm.pdf(z_a)
```

```
                                                             - (1/Z)*(norm.pdf(
                                                            z_a) - norm.pdf(z_b))
                                                            **2)

            delta_tau = (1/var_hat) - tau_cav - tau[i]
            tau[i] = tau[i] + delta_tau

            v[i] = (mu_hat/var_hat) - v_cav
            temp_sig = Sigma[:,i].reshape(-1, n)

            Sigma = Sigma - (temp_sig.T @ temp_sig)*(delta_tau**-1 +
                                                            Sigma[i,i])**-1

            mu = Sigma @ v

            self.Sigma = Sigma
            self.mu = mu

        elif data_types[i] == "point":

            cur_data = all_data[i]

            y = cur_data[self.dim]
            tau_cav = Sigma[i, i] ** -1 - tau[i]
            v_cav = mu[i] * (Sigma[i, i] ** -1) - v[i]

            mu_cav = v_cav / tau_cav
            sig_cav = 1 / tau_cav

            mu_hat = (sig_cav*y + self.noise*mu_cav)/(sig_cav + self.
                                                            noise)
            var_hat = 1/((1/sig_cav) + (1/self.noise))
            Z = (2*3.14159*var_hat)**0.5

            delta_tau = (1 / var_hat) - tau_cav - tau[i]


            tau[i] = tau[i] + delta_tau


            v[i] = (mu_hat / var_hat) - v_cav
            temp_sig = Sigma[:, i].reshape(-1, n)

            Sigma = Sigma - (temp_sig.T @ temp_sig) * (delta_tau ** -1
                                                             + Sigma[i, i]) ** -1

            mu = Sigma @ v

            self.Sigma = Sigma
            self.mu = mu
        elif data_types[i] == "ordinal":

            y = int(all_data[i][self.dim])
            t_low = self.b[y-1]
            t_high = self.b[y]


            tau_cav = Sigma[i, i] ** -1 - tau[i]
            v_cav = mu[i] * (Sigma[i, i] ** -1) - v[i]

            mu_cav = v_cav / tau_cav
            sig_cav = 1 / tau_cav


            z_a = (mu_cav - t_low) / ((self.noise ** (1 / 2)) * (1 + (
                                                    sig_cav / self.noise)
                                                    ) ** (1 / 2))

            z_b = (mu_cav - t_high) / ((self.noise ** (1 / 2)) * (1 +
                                                    (sig_cav / self.noise
                                                    )) ** (1 / 2))

            Z = norm.cdf(z_a) - norm.cdf(z_b)

            mu_hat = mu_cav + ((sig_cav / (Z * (self.noise ** 0.5))) *
                                                     (1 / (1 + (sig_cav /
                                                     self.noise)) ** 0.5)
                                                    ) * (norm.pdf(z_a) -
                                                    norm.pdf(z_b))

            var_hat = sig_cav + (sig_cav ** 2) * (1 / (Z * (self.noise
                                                     + sig_cav))) * (z_b
                                                    * norm.pdf(z_b) - z_a
                                                     * norm.pdf(z_a) - (1
                                                     / Z) * (norm.pdf(z_a
                                                    ) - norm.pdf(z_b)) **
                                                     2)

            delta_tau = (1 / var_hat) - tau_cav - tau[i]
            tau[i] = tau[i] + delta_tau

            v[i] = (mu_hat / var_hat) - v_cav
            temp_sig = Sigma[:, i].reshape(-1, n)

            Sigma = Sigma - (temp_sig.T @ temp_sig) * (delta_tau ** -1
                                                     + Sigma[i, i]) ** -1

            mu = Sigma @ v

            self.Sigma = Sigma
```

```
                self.mu = mu


        if (sum(np.square(mu - prev_mu))**0.5)/n < 0.001:

            converged = True

    self.mu = mu
    self.tau = tau

def predict_mean(self, x):

    x = np.array(x).reshape(-1, self.dim)

    k_star = self.kernel(x, np.array(self.all_x))
    K = self.kernel(np.array(self.all_x), np.array(self.all_x))

    S = self.tau*np.identity(len(self.tau))

    v = S @ self.mu


    mean = k_star @ inv((K + inv(S))) @ inv(S) @ v


    return mean

def predict_var(self, x):

    x = np.array(x).reshape(-1, self.dim)
    k_star_star = self.kernel(x, x)

    k_star = self.kernel(x, np.array(self.all_x))

    K = self.kernel(np.array(self.all_x), np.array(self.all_x))
    S = self.tau * np.identity(len(self.tau))

    var = k_star_star - k_star @ inv(K + inv(S)) @ k_star.T

    return var.diagonal()

def predict_Sigma(self, x):

    x = np.array(x).reshape(-1, self.dim)
    k_star_star = self.kernel(x, x)

    k_star = self.kernel(x, np.array(self.all_x))

    K = self.kernel(np.array(self.all_x), np.array(self.all_x))
    S = self.tau * np.identity(len(self.tau))
    var = k_star_star - k_star @ inv(K + inv(S)) @ k_star.T

    return var

def likelihood(self, y, f):

    z1 = (self.b[y]-f)/(self.n_var**(1/2))
    z2 = (self.b[y-1]-f)/(self.n_var**(1/2))

    return norm.cdf(z1) - norm.cdf(z2)

def visualise_GP_2D(self):

    all_x_unrolled = [a[0] for a in self.all_x]
    min_x = min(all_x_unrolled)
    max_x = max(all_x_unrolled)

    viz_grid = np.linspace(min_x, max_x, num=30)
    mesh = get_mesh(viz_grid)

    mu = self.predict_mean(mesh)

    ax = plt.axes(projection='3d')
    ax.scatter3D(mesh[:, 0], mesh[:, 1], mu, c=mu, cmap='viridis')
    plt.show()

def visualise_GP(self, viz_train_data=False, viz_variance=False):  # one
                                         dimensional

    assert self.dim == 1

    all_x_unrolled = [a[0] for a in self.all_x]
    min_x = min(all_x_unrolled)
    max_x = max(all_x_unrolled)

    viz_grid = np.linspace(min_x, max_x, num=500)

    predict_mu = self.predict_mean(viz_grid).reshape(-1)
    plt.plot(viz_grid, predict_mu, color='blue')

    if viz_variance:

        predict_var = self.predict_var(viz_grid) - self.noise**2
        std = predict_var**0.5


        plt.fill_between(viz_grid.reshape(-1, ), predict_mu-2*std,
                                                     predict_mu+2*std, alpha=0.6)

    plt.show()
```

8.2 Data Generation

Below we give the relevant code for generating artificial data in 1,2 and 3 dimensions.

```
from numpy import linspace
from numpy.random import uniform, choice
from scipy.stats import norm
from matplotlib import pyplot as plt
import numpy as np
from scipy.spatial import distance_matrix
from mpl_toolkits import mplot3d

def artificial_data(low=0, high=10, interval_num=25, point_num=25,
                                    ordinal_num=25, grid_size=250,
                                    thresh_num=3):

    n = interval_num + point_num + ordinal_num

    # draw from a gaussian process prior w/ mean 0 and some random
                                          reasonable kappa

    l = 0.2
    n_std = float(1.5*np.random.rand(1))
    x_ = np.linspace(low, high, grid_size).reshape(-1,1)
    n_std=0.1
    distances = np.square(distance_matrix(x_, x_))

    Sigma = np.exp(-distances / (2*l**2))

    gauss_draw = np.random.multivariate_normal(np.zeros(grid_size), Sigma)

    plt.plot(x_, gauss_draw)
    plt.show()

    # choose random points to generate from (first a are interval, next b
                                          point, last c ordinal)

    rand_points = np.random.choice(grid_size, n)
    mus = gauss_draw[rand_points]

    # generate intervals

    intervals = [[float(x_[rand_points[i]]), generate_interval(mu, n_std)[
                                          0:2]] for i, mu in enumerate(mus[
                                          0:interval_num])]

    for i in range(len(intervals)):

        intervals[i] = [intervals[i][0], intervals[i][1][0], intervals[i][
                                              1][1]]

    # generate points
    points = np.random.normal(mus[interval_num:interval_num+point_num],
                                     scale=n_std*np.ones(point_num))
    points = [[float(x_[rand_points[i+interval_num]]), p] for i, p in
                                     enumerate(points)]

    # generate ordinals

    quantiles = np.quantile(gauss_draw, [i/thresh_num for i in range(1,
                                     thresh_num)])
    print(quantiles)

    ordinal = [[float(x_[rand_points[i+interval_num+point_num]]),
                                     generate_ordinal(mu, n_std,
                                     quantiles)] for i, mu in
                                     enumerate(mus[interval_num+
                                     point_num:n])]

    #ordinal = [o for o in ordinal if o[1] not in [1, thresh_num]]
    f = gauss_draw

    return intervals, points, ordinal, n_std, l, f, x_, quantiles, Sigma


def artificial_data2d(low=0, high=10, interval_num=25, point_num=25,
                                 ordinal_num=25, grid_size=250,
                                 thresh_num=3):

    n = interval_num + point_num + ordinal_num

    # draw from a gaussian process prior w/ mean 0 and some random
                                     reasonable kappa

    l = 0.6
    n_std = float(1.5*np.random.rand(1))
    x_ = np.linspace(low, high, grid_size).reshape(-1,1)
    x,y = np.meshgrid(x_,np.linspace(low, high, grid_size).reshape(-1,1),
                                     copy=False)
    x_points = np.vstack([x.reshape(-1), y.reshape(-1)]).reshape(-1,2)
    n_std=0.1

    x_points = get_mesh(x_)
    distances = np.square(distance_matrix(x_points, x_points))

    Sigma = np.exp(-distances / (2*l**2))
    gauss_draw = np.random.multivariate_normal(np.zeros(x_points.shape[0])
                                     , Sigma)

    ax = plt.axes(projection='3d')
    ax.scatter3D(x_points[:,0], x_points[:,1], gauss_draw, c=gauss_draw,
                                     cmap='viridis')
    plt.show()

    # choose random points to generate from (first a are interval, next b
                                            point, last c ordinal)

    rand_points = np.random.choice(grid_size**2, n)
    mus = gauss_draw[rand_points]

    # generate intervals

    intervals = [[list(x_points[rand_points[i], :]), generate_interval(mu,
                                             n_std)[0:2]] for i, mu in
                                            enumerate(mus[0:interval_num])]

    for i in range(len(intervals)):

        intervals[i] = [intervals[i][0][0], intervals[i][0][1], intervals[
                                                i][1][0], intervals[i][1][1]]

    # generate points

    points = np.random.normal(mus[interval_num:interval_num+point_num],
                                            scale=n_std*np.ones(point_num))
    points = [[list(x_points[rand_points[i+interval_num], :]), p] for i, p
                                             in enumerate(points)]

    for i in range(len(points)):
        points[i] = [points[i][0][0], points[i][0][1], points[i][1]]

    # generate ordinals

    quantiles = np.quantile(gauss_draw, [i/thresh_num for i in range(1,
                                            thresh_num)])

    ordinal = [[list(x_points[rand_points[i+interval_num+point_num], :]),
                                            generate_ordinal(mu, n_std,
                                            quantiles)] for i, mu in
                                            enumerate(mus[interval_num+
                                            point_num:n])]
    for i in range(len(ordinal)):
        ordinal[i] = [ordinal[i][0][0], ordinal[i][0][1], ordinal[i][1]]
    #ordinal = [o for o in ordinal if o[1] not in [1, thresh_num]]
    f = gauss_draw

    return intervals, points, ordinal, n_std, l, f, x_, x_points,
                                            quantiles, Sigma

def artificial_data3d(low=0, high=10, interval_num=25, point_num=25,
                                        ordinal_num=25, grid_size=250,
                                        thresh_num=3):
```

```
n = interval_num + point_num + ordinal_num


l = 0.6
n_std = float(1.5*np.random.rand(1))
x_ = np.linspace(low, high, grid_size).reshape(-1,1)
x,y = np.meshgrid(x_,np.linspace(low, high, grid_size).reshape(-1,1),
                                      copy=False)
x_points = np.vstack([x.reshape(-1), y.reshape(-1)]).reshape(-1,2)
n_std=0.1

x_points = get_mesh3d(x_)
distances = np.square(distance_matrix(x_points, x_points))

Sigma = np.exp(-distances / (2*l**2))
gauss_draw = np.random.multivariate_normal(np.zeros(x_points.shape[0])
                                      , Sigma)

# choose random points to generate from (first a are interval, next b
                                      point, last c ordinal)

rand_points = np.random.choice(grid_size**3, n)
mus = gauss_draw[rand_points]

# generate intervals

intervals = [[list(x_points[rand_points[i], :]), generate_interval(mu,
                                       n_std)[0:2]] for i, mu in
                                      enumerate(mus[0:interval_num])]

for i in range(len(intervals)):

    intervals[i] = [intervals[i][0][0], intervals[i][0][1], intervals[
                                         i][0][2], intervals[i][1][0],
                                          intervals[i][1][1]]

# generate points

points = np.random.normal(mus[interval_num:interval_num+point_num],
                                      scale=n_std*np.ones(point_num))
points = [[list(x_points[rand_points[i+interval_num], :]), p] for i, p
                                       in enumerate(points)]

for i in range(len(points)):
    points[i] = [points[i][0][0], points[i][0][1], points[i][0][2],
                                         points[i][1]]

# generate ordinals

quantiles = np.quantile(gauss_draw, [i/thresh_num for i in range(1,
                                      thresh_num)])
    ordinal = [[list(x_points[rand_points[i+interval_num+point_num], :]),
                                          generate_ordinal(mu, n_std,
                                          quantiles)] for i, mu in
                                          enumerate(mus[interval_num+
                                          point_num:n])]
    for i in range(len(ordinal)):
        ordinal[i] = [ordinal[i][0][0], ordinal[i][0][1], ordinal[i][0][2]
                                              ,ordinal[i][1]]

    f = gauss_draw

    return intervals, points, ordinal, n_std, l, f, x_, x_points,
                                          quantiles, Sigma

def generate_ordinal(mu, std, thresh):  # thresh = thresholds

    point = np.random.normal(mu, std)

    thresh = [-10000] + list(thresh) + [10000]

    for i in range(len(thresh)):

        if thresh[i-1] <= point < thresh[i]:

            return i

def generate_interval(mu, sig, l_bounds=None):

    # given a normal distribution, return a sampled interval

    if not l_bounds:

        l_bounds = [6*sig, 6*sig]

    l = uniform(l_bounds[0], l_bounds[1])

    grid = linspace(mu-3*sig, mu+3*sig-l, num=300)  # generates the a in [
                                          a,b] ([a,b] = [a,b+l])

    probs = [norm.cdf(a+l, loc=mu, scale=sig) - norm.cdf(a, loc=mu, scale=
                                          sig) for a in grid]
    probs = probs/sum(probs)

    a = choice(grid, p=probs)

    return [a,a+l]

def get_mesh(x):

    mesh = []
    for i, p in enumerate(x):

        for j, q in enumerate(x):

            mesh.append([p,q])

    return np.array(mesh).reshape(-1,2)

def get_mesh3d(x):

    mesh = []
    for i, p in enumerate(x):

        for j, q in enumerate(x):

            for k, r in enumerate(x):

                mesh.append([p,q,r])

    return np.array(mesh).reshape(-1,3)
```

8.3 BALD functions

Below we give the main functions used for approximation of BALD quantities.

```
def kl_mvn(m0, S0, m1, S1):
    """
    below code is based on stack overflow answer at
    https://stackoverflow.com/questions/44549369/kullback-leibler-
                                                divergence-from-gaussian-pm-pv-to
                                                -gaussian-qm-qv
    """
    # store inv diag covariance of S1 and diff between means
    N = m0.shape[0]
    iS1 = np.linalg.inv(S1)
    diff = m1 - m0

    d = S0[0, 0]
    G = cholesky(S1).diagonal()

    tr_term = np.trace(iS1 @ S0)
    det_term = 2 * np.sum(np.log(G)) - N * np.log(d)
    quad_term = diff.T @ np.linalg.inv(S1) @ diff   # np.sum( (diff*diff) *
                                                 iS1, axis=1)


    return float(.5 * (tr_term + det_term + quad_term - N))


def compute_BALD_point_data(Zp, interval, s1, mup, c):
    nsamples = Zp.shape[1]  # Zp here is a [num_x_points, nsamples] of
                                        samples from posterior
    BALD = []
    for i in range(mup.shape[0]):
        p = np.vstack(
            [norm.cdf((interval[1:] - Zp[i, j]) / s1) - norm.cdf((interval
                                                      [:-1] - Zp[i, j]) / s1)
                                                      for j in
             range(nsamples)])
        # p = np.minimum(1-1e-9,np.maximum(p,1e-9))#cut to avoid NAN in
                                                  the log
        p1 = np.mean(p, axis=0)
        entropy1 = np.sum(np.nan_to_num(-p1 * np.log2(p1), 0)) / (np.log2(
                                                  len(interval) - 1))
        entropy2 = np.sum(np.nan_to_num(np.mean(-p * np.log2(p), axis=0),
                                                  0)) / (np.log2(len(interval)
                                                  - 1))
        BALD.append((entropy1 - entropy2)*(np.log2(len(interval) - 1))/c)
    return BALD


def compute_BALD_interval_data(Zp, num_intervals, l, s1, mup, c):
    nsamples = Zp.shape[1]  # Zp here is a [num_x_points, nsamples] of
                                        samples from posterior, l here is
                                         the length of the intervals
    BALD = []

    # get the intervals first, we just get the start points
    interval_start = np.array([x for x in list(np.linspace(-2, 2-l,
                                        num_intervals))])
    interval_end = np.array([x+l for x in list(np.linspace(-2, 2-l,
                                        num_intervals))])
    unif = len(interval_start)
    for i in range(mup.shape[0]):
        p = np.vstack(
            [norm.cdf((interval_end - Zp[i, j]) / s1) - norm.cdf((
                                                      interval_start - Zp[i, j]
                                                      ) / s1) for j in
             range(nsamples)])

        p = p/np.sum(p, axis=1).reshape(-1,1)
        p = np.nan_to_num(p, 0)
        # p = np.minimum(1-1e-9,np.maximum(p,1e-9))#cut to avoid NAN in
                                                  the log
        p1 = np.mean(p, axis=0)

        entropy1 = np.sum(np.nan_to_num(-p1 * np.log2(p1), 0)) / (np.log2(
                                                  unif - 1))
        entropy2 = np.sum(np.nan_to_num(np.mean(-p * np.log2(p), axis=0),
                                                  0)) / (np.log2(unif - 1))
        BALD.append((entropy1 - entropy2)*(np.log2(unif - 1))/c)
    return BALD

def compute_BALD_ordinal_data(Zp,thresholds,s1, mup, c):
  nsamples = Zp.shape[1]
  BALD=[]
  interval = np.array([-np.inf] + list(thresholds) + [np.inf])
  for i in range(mup.shape[0]):
    p= np.vstack([norm.cdf((interval[1:]-Zp[i,j])/s1)-norm.cdf((interval[:
                                              -1]-Zp[i,j])/s1) for j in range(
                                              nsamples)])
    #p = np.minimum(1-1e-9,np.maximum(p,1e-9))#cut to avoid NAN in the log
    p1 =np.mean(p,axis=0)
    entropy1 = np.sum(np.nan_to_num(-p1*np.log2(p1),0))/(np.log2(len(
                                              interval)-1))
    entropy2 = np.sum(np.nan_to_num(np.mean(-p*np.log2(p),axis=0),0))/(np.
                                              log2(len(interval)-1))
    BALD.append((entropy1-entropy2)*np.log2(len(interval)-1)/c)
  return BALD


def max_BALD(num_samples, pred_mu, pred_Sigma, noise, thresholds,
                                        interval_length, x_points, costs):

    # calculate BALD quantity across grid for each data type, return point
                                               to query + type of data to query
    # posterior samples
    M = len(x_points)
    Zp = np.random.multivariate_normal(pred_mu[:, 0], pred_Sigma,
                                              num_samples).T

    BALD_point = compute_BALD_point_data(Zp, np.linspace(-3,3,100), noise,
                                               pred_mu, costs["point"])
    BALD_interval = compute_BALD_interval_data(Zp, 100, interval_length,
                                              noise, pred_mu, costs["interval"]
                                              )
    BALD_ord = compute_BALD_ordinal_data(Zp, thresholds, noise, pred_mu,
                                              costs["ordinal"])

    max_idx = int(np.argmax(BALD_point + BALD_interval + BALD_ord))

    dtype = None

    if max_idx < M:
        dtype = "point"
    if M <= max_idx < 2*M:
        dtype = "interval"
    if 2*M<= max_idx < 3*M:
        dtype= "ordinal"
    max_idx = max_idx % M
    x = x_points[max_idx]
    return x, dtype, max_idx

def max_BALD_norm(num_samples, pred_mu, pred_Sigma, noise, thresholds,
                                    interval_length, x_points, costs):

    # calculate BALD quantity across grid for each data type, return point
                                          to query + type of data to query
    # posterior samples
    M = len(x_points)
    Zp = np.random.multivariate_normal(pred_mu[:, 0], pred_Sigma,
                                         num_samples).T

    BALD_point = np.array(compute_BALD_point_data(Zp, np.linspace(-3,3,300
                                         ), noise, pred_mu, costs["point"]
                                         ))*costs["point"]/np.log2(299)
    BALD_interval = np.array(compute_BALD_interval_data(Zp, 100,
                                         interval_length, noise, pred_mu,
                                         costs["interval"]))*costs["
                                         interval"]/np.log2(99)
    BALD_ord = np.array(compute_BALD_ordinal_data(Zp, thresholds, noise,
                                         pred_mu, costs["ordinal"]))*costs
                                         ["ordinal"]/np.log2(3)

    max_idx = int(np.argmax(BALD_point + BALD_interval + BALD_ord))

    dtype = None

    if max_idx < M:
        dtype = "point"
    if M <= max_idx < 2*M:
        dtype = "interval"
    if 2*M<= max_idx < 3*M:
        dtype= "ordinal"
    max_idx = max_idx % M
    x = x_points[max_idx]

    return x, dtype, max_idx
```

8.4 Example

We give here a simple example of some artificial data being generated. 2 points, 2 ordinals and 2 intervals are sampled randomly from the data. A GP object is created and uses EP to find a posterior according to these data.

```python
import numpy as np
import generate_data
import GaussianProcess

intervals, points, ord, noise, l, f, x, b, Sigma = generate_data.
                                        artificial_data2d(0, 1, interval_num=
                                        2, point_num=2, ordinal_num=2,
                                        grid_size=30, thresh_num=3)

gp = GaussianProcess(dim=2,l=l, noise=noise, amplitude=1, b=list(b))

points = np.array(points)
intervals = np.array(intervals)
ord = np.array(ord)
gp.add_train_data(points, data_type="point")
gp.add_train_data(intervals, data_type="interval")
gp.add_train_data(ord, data_type="ordinal")

gp.Full_EP()
gp.visualise_GP_2D()
```