8th ICML Workshop on Automated Machine Learning (2021)

GPy-ABCD: A Configurable Automatic Bayesian Covariance Discovery Implementation

T. Fletcher T.Fletcher-6@sms.ed.ac.uk
The University of Edinburgh

A. Bundy A.Bundy@ed.ac.uk
The University of Edinburgh

K. Nuamah K.Nuamah@ed.ac.uk
The University of Edinburgh

Abstract

Gaussian Processes (GPs) are a very flexible class of nonparametric models frequently used in supervised learning tasks because of their ability to fit data with very few assumptions, namely just the type of correlation (kernel) the data is expected to display. Automatic Bayesian Covariance Discovery (ABCD) is an iterative GP regression framework aimed at removing the requirement for even this initial correlation form assumption. An original ABCD implementation exists and is a complex stand-alone system designed to produce long-form text analyses of provided data. This paper presents a lighter, more functional and configurable implementation of the ABCD idea, outputting only fit models and short descriptions: the Python package GPy-ABCD, which was developed as part of an adaptive modelling component for the FRANK query-answering system. It uses a revised model-space search algorithm and removes a search bias which was required in order to retain model explainability in the original system.

Introduction

Automatic Bayesian Covariance Discovery (ABCD) (Lloyd, James Robert et al., 2014) is an unsupervised learning system acting as a framework which iteratively runs Gaussian Process (GP) regressions in order to select the best fitting model within some functional-form search-space limits. Its noteworthy utility lies not just in the generally close fit one can expect from it, but in the high level of interpretability its outputs carry, being able to describe even functional forms which vary over the given domain. An example for clarity: a pattern which an ABCD system is able to identify and describe in text would be (paraphrasing to shorten it) "the data starts as a linear function but then begins a periodic exponential growth". The original ABCD implementation (now the main constituent of the Automatic Statistician (Steinruecken, Christian et al., 2019)), is a large project with parts written in MATLAB, Python, Mathematica, Fortran and more, and the outputs it produces are very detailed multi-page, text-based analyses of the given data, describing contributions from each component of the identified modular functional form. This paper describes a more functional and highly configurable ABCD system implementation which focusses on improving the model-space search and not on the output analysis, instead returning just the models and 1-paragraph descriptions. GPy-ABCD is an open-source Python package (Fletcher, 2020) built on the GPy library developed by the Sheffield machine learning group (2012); it is intended as a standard modelling tool to be used in a broader analysis

©2021 T. Fletcher, A. Bundy and K. Nuamah.

workflow; in particular, the workflow it was developed for is an adaptive statistical modelling component for the Functional Reasoning for Acquiring Novel Knowledge (FRANK) Query-Answering (QA) system (Nuamah et al., 2016; Bundy et al., 2018).

1. Background

Grosse, Roger et al. (2012) noted (as others had before) that many common probabilistic models can be represented as compositions of simpler ones, and they used the class of matrix decomposition models to construct a context-free grammar generating a compositional model space. Duvenaud, David et al. (2013) built on this work and created the proof-of-concept which later became the ABCD system (Lloyd, James Robert et al., 2014). Extensions to this research include joint data modelling (separating common features from individual ones) (Tong and Choi, Jaesik; Hwang, Yunseong and Choi, Jaesik, 2015), model criticism (Lloyd, James Robert and Ghahramani, Zoubin, 2015) and Bayesian corrections to final-model variance (Janz, David et al., 2016), while the original ABCD became part of the Automatic Statistician (Riaz Moola; Kim, Hyunjik and Teh, Yee Whye, 2017; Steinruecken, Christian et al., 2019), also exploring uses in classification tasks (Nikola Mrkšić). GPy-ABCD is based on the Automatic Statistician's ABCD and was designed as a simple and developer-friendly modelling tool since its functionality was required within the FRANK QA system (Nuamah et al., 2015, 2016; Nuamah and Bundy, 2019, 2018; Bundy et al., 2018). Briefly, FRANK fits into the so-called "Third wave of AI", employing both symbolic and statistical reasoning to answer data-focussed queries, e.g. "Will the African country with highest GDP in 2040 have a higher population than the equivalent South-American one?", which requires language parsing, online data sourcing, symbolic reasoning and statistical modelling to answer. In particular, GPy-ABCD was created as part of FRANK's adaptive statistical modelling component Statistical Methodology Advisor at Reasoning Time (SMART), whose purpose is to select and execute specific statistical methodologies to match specific query and data properties. In this context GPy-ABCD would be selected in place of other statistical methods for queries involving functional shape description (e.g. "How does rainfall in the UK behave over time?" or "Is population growth in Asia periodic/linear/-exponential?"). GPy-ABCD is concerned with re-implementing only the simplest-output ABCD functionality, as that is what is required by FRANK. Comparing the respective broader systems, i.e. the Automatic Statistician and FRANK, the former is concerned with producing in-depth analyses of directly provided data of specific kinds, while the latter is a general-purpose QA system automating data procurement and method choice.

2. The ABCD Idea

Parametric regression methods are defined by strict assumptions on the nature of the data being analysed, i.e. the form of the predicting function the parameters of which they tune (e.g. in the case of polynomials, the assumption is the power up to which to model). Nonparametric models on the other hand place much lower restrictions on the predicting function's form, using the data itself to adjust it, making them ideal candidates for learning tasks. Standard examples of nonparametric modelling methods are Support Vector Machines, GPs and different variations of Splines, all of which still require some initial

assumptions to restrict their functional forms. In the case of GPs (briefly explained in Appendix A) the assumption is the choice of covariance function the data is expected to exhibit, and although the generality of the common kernels allows extreme fitting flexibility, what is lost in exchange is a level of data interpretability, making kernel choice still subject to the modeller's judgement prior to fitting. ABCD is an iterative GP regression framework which explores a space of modularly-constructed kernels in order to identify the ones which best balance closeness of fit and expression complexity, thus reducing the required modelling assumptions even further. As laid out in (Lloyd, James Robert et al., 2014), the core components of this framework are the following:

1. An open-ended and expressive language of models
2. An efficient generation and search procedure to explore the model space
3. A model evaluation and comparison method balancing complexity and closeness of fit
4. A procedure to automatically generate full reports of the best candidate

3. Implementation

The following sub-sections cover GPy-ABCD's components as outlined above; they prioritise abstract description over specific details since each section is mirrored by one in Appendix B, where differences from the original ABCD are also explained.

3.1 The Model Language

The model language is defined by a context-free grammar constituted by a specific selection of base GP kernels which can be combined by direct addition and multiplication to produce more complex modular kernels. The criterion by which base kernels are selected is to cover data features which are common and simple enough to be easily interpretable, resulting in the following set (Duvenaud, David et al., 2013) (details in Section B.1):

- White Noise (WN) kernel to model uncorrelated noise
- Constant (C) kernel to model constant functions (useful for simple mean shifts)
- Linear (LIN) kernel to model linear functions and, when repeatedly multiplied by itself, higher order polynomials
- Squared Exponential (SE) kernel to model generic smooth functions
- (Generalised) Periodic (PER) kernel to model generic periodic functions

Though present and usable, the SE kernel is disabled by default in GPy-ABCD since its versatility and small number of parameters make it too competitive against more complex but more interpretable expressions. Figure 1 shows examples of curves from a simple multiplication of kernel expressions. In addition to the above, the grammar contains two more composite kernels built on lower-level sigmoidal kernels (see Appendix B.1):

- Change-Point (CP) operator: $CP[k_1, k_2] = S \times k_1 + Sr \times k_2$
- Change-Window (CW) operator: $CW[k_1, k_2] = SI \times k_1 + SIr \times k_2$

These allow the inclusion in the language of models which permanently or temporarily change covariance form, e.g. from linear to periodic (examples in Figure 2).

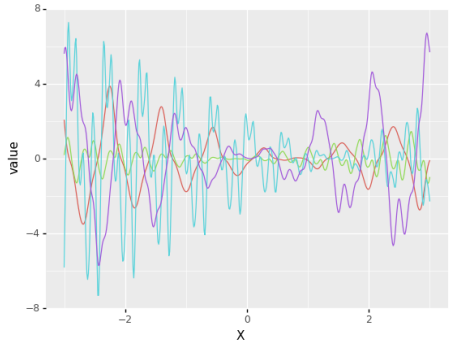

Figure 1: $LIN \times PER$: periodic functions with linearly varying amplitudes

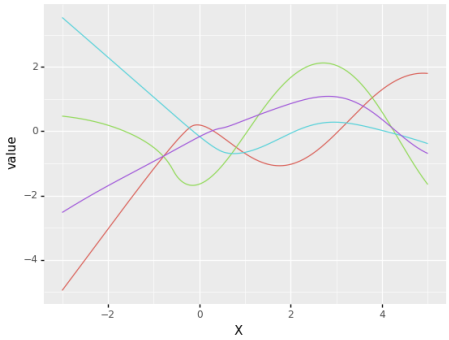

Figure 2: $CP(LIN, PER)$: linear functions becoming periodic

3.2 Kernel Expressions & Simplification

The core of GPy-ABCD is the `KernelExpression` (abstract) class, which is the symbolic representation of kernels: a non-trivial kernel is represented by a tree of `KernelExpression` nodes which represent kernel operations; the base kernel arguments are contained directly in the node, while more complex arguments are linked to (children nodes). For example $LIN \times (PER + C)$ has two nodes: a root containing the factor LIN and a child containing the sum of PER and C. The kernel expression classes have methods providing the following general functionality:

- They self-simplify when modified
- They can produce GPy kernel objects
- They can extract the fit model parameters from a matching GPy object
- They can rearrange to a (canonical) sum-of-products form
- They can generate text descriptions of their sum-of-products form

Simplification of Kernel Expressions (KEs) is just mathematical rearrangement into more succinct forms by some basic rules (see Section B.2). Here are two simplification examples for clarity: $LIN \times PER \times WN \rightarrow LIN \times WN$ and $(SE \times SE + LIN) \times C \rightarrow SE + LIN$. These simplifications are self-triggering and take place both before model fitting and when later reducing to canonical sum-of-products form.

3.3 The Model-Space Search

GPy-ABCD's model-space search algorithm is a key difference from the original ABCD; it is essentially a beam search using a context-free grammar as the successor states' generator (and beam search is essentially a limited-bandwidth best-first-search). The overall algorithm is visualised in Figure 3 and explained below, with the main configurable inputs underlined; a more detailed description making reference to all inputs to the main search function in the library is provided in Section B.3. After fitting an initial (heuristic) list of simple KEs, a predetermined number (M) of search rounds is performed; each round consists of the expansion (by grammar production rules) of the N best-fitting KEs so far, which are then

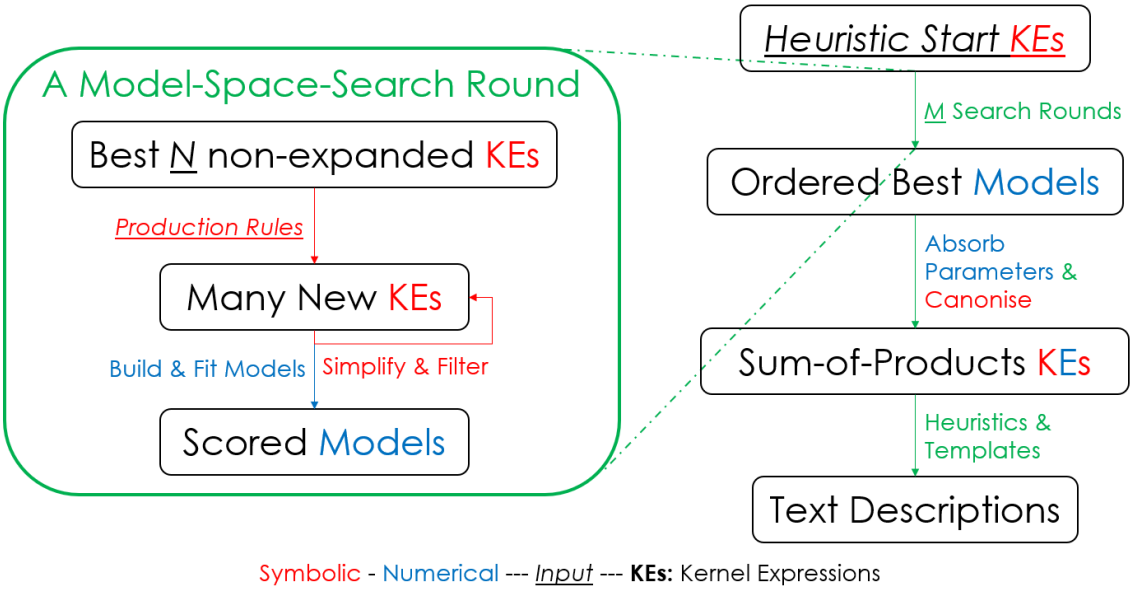

Figure 3: The full model-space search algorithm

simplified and filtered for duplicates before fitting (the simplified KEs do not need to be in sum-of-products form in order to retain final model interpretability, which was a limitation of the original ABCD; see Section B.2). Each model is scored by a given utility function which balances closeness of fit with expression complexity (see Section B.3.1 for a discussion on possibilities). The main algorithm output is then the ordered list of best fitting models.

3.4 Model Description

By converting a fit kernel to its canonical sum-of-product form, each product's factors are ordered by a fixed pattern: PER, WN, SE, C, LIN and finally any sigmoidal kernel. This is done in order to assign fixed roles to specific kernels if present in particular combinations; the first available kernel takes the role of principal functional form and all the others of its modifiers. The sentence production is then performed by simple templates which describe each kernel differently according to its role and fill in parameter values. Tables detailing these combinations are available in the ABCD literature (Lloyd, James Robert et al., 2014), but as an example, an example output for $LIN \times (PER + C)$ would be:

> "The fit kernel consists of 2 components:
> - A linear function with offset -0.09; this component has overall variance 1.04
> - A periodic function with period 6.24 and lengthscale 1774.03, with linearly varying amplitude with offset -0.09; this component has overall variance 0.54"

where the "2 components" are from distributing the product. This system works well because the base kernels were chosen from the beginning with the purpose of interpretability.

4. Evaluation

The main hypothesis under evaluation for GPy-ABCD is twofold:

- That it recognises the correct patterns in synthetic data as an ABCD system should (i.e. that its kernels do indeed work individually and together)
- That it fits and describes complex data similarly enough (taking into account the algorithm difference) to the original ABCD system implementation

Separately, it needs to fulfil the role it was originally designed for in the FRANK QA system, which means being able to provide the data required to introduce the new types of queries mentioned in Section 1. Appendix C expands on the below summary.

The 1st point was evaluated by trying to recover known kernels when fitting noisy data produced from a set of formulae of varying complexities (e.g. $LIN \times (PER + C)$ is the "obvious" match for $y = 2x\cos(\frac{x-5}{2})^2$). The system identified the intended kernel in all cases (though not necessarily in the top result) except for the class of CW kernels with non-stationary first-arguments (e.g. $CW(LIN, \cdots)$); see Section C.1 for a discussion on the reasons. The 2nd point was evaluated by trying to replicate the core outputs of the original ABCD system on the datasets in their literature. Although no noticeable differences in closeness of fit were present, the KEs identified by the two systems were broadly similar but not matching, with GPy-ABCD's expressions being generally simpler than the original's; this is reasonable since the comparison setup tried to match the number of rounds of the respective algorithms and not their search depth (see Section C.2), but the effects of the differences in implementation and underlying frameworks and fitting libraries are an unquantified factor. With regards to the last point of evaluation, GPy-ABCD is indeed able to provide FRANK with the means to implement the target functionality, and some useful infrastructure was added to both projects in order to manage computation times (since ABCD is a very computationally expensive method in a QA context).

Conclusion

GPy-ABCD is a working implementation of an ABCD modelling system, replicating and improving the core components of the original one (i.e. not the production of detailed analyses); comparing the two, GPy-ABCD stands out for the improved model-search algorithm and extensive customisability. Though there is room for further numerical method improvements and investigation of search path behaviour differences with the original implementation, the system performs well on both synthetic and real data, and can help users identify and describe patterns in the exploratory data analysis phase of research. Partly due to having been developed to serve as one of many components in a broader adaptive-modelling system, GPy-ABCD constitutes a solid base on which to build further functionality and research: the identified issue of statistical appropriateness of the choice of utility function (Section B.3.1) is the primary theory-based next step, while more practical expansion directions include extending the system to handle multidimensional data and adding more fitting methods (two things which GPy is already equipped for), as well as further increasing user customisation to encompass providing arbitrary base kernels, since specific scenarios (including dimension-based) may require broader or narrower selections to work well or for optimisation purposes.

References

Automatic model construction with Gaussian processes. PhD thesis, Duvenaud, David, 2014.

Tomohiro Ando. Bayesian predictive information criterion for the evaluation of hierarchical bayesian and empirical bayes models. *biometrika*, 94(2):443–458, 2007. doi: 10.1093/BIOMET/ASM017.

Alan Bundy, Kwabena Nuamah, and Christopher Lucas. Automated reasoning in the age of the internet. pages 3–18, 2018. doi: 10.1007/978-3-319-99957-91.

Duvenaud, David, James Lloyd, Grosse, Roger, Joshua Tenenbaum, and Ghahramani Zoubin. Structure discovery in nonparametric regression through compositional kernel search. pages 1166–1174, 2013.

Thomas Fletcher. Gpy-abcd, 2020. URL `https://github.com/T-Flet/GPy-ABCD`.

Grosse, Roger, Salakhutdinov, Ruslan R, Freeman, William T, and Tenenbaum, Joshua B. Exploiting compositionality to explore a large space of model structures, 2012.

Hwang, Yunseong and Choi, Jaesik. The automatic statistician: A relational perspective, 2015.

Janz, David, Paige, Brooks, Rainforth, Tom, Meent, Jan-Willem van de, and Wood, Frank D. Probabilistic structure discovery in time series data, 2016.

Kim, Hyunjik and Teh, Yee Whye. Scaling up the automatic statistician: Scalable structure discovery using gaussian processes, 2017.

Sadanori Konishi and Genshiro Kitagawa. *Information Criteria and Statistical Modeling*. 2007.

Lloyd, James Robert and Ghahramani, Zoubin. Statistical model criticism using kernel two sample tests. pages 829–837, 2015.

Lloyd, James Robert, David Kristjanson Duvenaud, Roger Baker Grosse, Joshua B. Tenenbaum, and Ghahramani, Zoubin. Automatic construction and natural-language description of nonparametric regression models. pages 1242–1250, 2014.

Sheffield machine learning group. Gpy, 2012. URL `https://sheffieldml.github.io/GPy/`.

Nikola Mrkšić. *Kernel Structure Discovery for Gaussian Process Classification*. PhD thesis, Computer Laboratory, University of Cambridge.

Kwabena Nuamah and Alan Bundy. Calculating error bars on inferences from web data. pages 618–640, 2018. doi: 10.1007/978-3-030-01057-748.

Kwabena Nuamah and Alan Bundy. Explainable inference in the frank question answering system. 2019.

Kwabena Nuamah, Alan Bundy, and Christopher Lucas. Using rich inference to find novel answers to questions, 2015.

Kwabena Nuamah, Alan Bundy, and Christopher Lucas. Functional inferences over heterogeneous data. pages 159–166, 2016. doi: 10.1007/978-3-319-45276-012.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge Mass., 2006. ISBN 10 0-262-18253-X. URL `http://www.gaussianprocess.org/gpml/chapters/`.

Riaz Moola. *Towards an Artificial Intelligence for Regression*. PhD thesis, Computer Laboratory, University of Cambridge.

Steinruecken, Christian, Smith, Emma, Janz, David, Lloyd, James Robert, and Ghahramani, Zoubin. The automatic statistician, 2019.

Anh Tong and Choi, Jaesik. Discovering latent covariance structures for multiple time series.

Aki Vehtari, Tommi Mononen, Ville Tolvanen, Tuomas Sivula, and Ole Winther. Bayesian leave-one-out cross-validation approximations for gaussian latent variable models. *journal of machine learning research*, 17(1):3581–3618, 2016.

Sumio Watanabe. A widely applicable bayesian information criterion. *journal of machine learning research*, 14(1):867–897, 2013.

Appendix A. Gaussian Process Regression

Definition 1 *(Rasmussen and Williams, 2006) A **Gaussian Process (GP)** is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

A GP f is fully defined by its mean function m and covariance ("kernel") function k ($f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'}))$), but given the Bayesian nature of the fitting process, specifically the fact that a mean-0 prior does not limit the posterior mean to 0 as well, it is common to assume the prior mean to be 0 and therefore let GPs be completely specified by their covariance function, which also simplifies notation. From the simple description above it should be clear that a GP regression does not fit an analytic expression for the given data but instead one for the process which could have produced it (accounting for noise): the tuned parameters are for the (covariance of the) distribution from which the input could have been sampled. Consequently predictions made from a GP model are not simple points or vectors with errors, but individual univariate or multivariate Gaussian distributions. $k(\boldsymbol{x}, \boldsymbol{x'})$ is usually also referred to as the GP's "kernel", and in typical GP regression scenarios the key feature a user is looking for in their kernel is generality, leading many applications to use the very flexible SE kernel (often also referred to as Radial Basis Function kernel): $k_{SE}(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2 \exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x'})^2}{2l^2}\right)$, where σ^2 is the variance and l is the lengthscale. Variance and lengthscale are parameters shared by many kernels, and intuitively, taking the kernel as a description of similarity between data observations, the variance regulates the average distance from the whole process mean, while the lengthscale regulates the average length of the fit curve's undulations (also serving as a gauge of reliable extrapolation distance). The above being granted, the choice of kernel is instead at times the crucial step in one's analysis in order to match the given data features and known context since it determines the generalisation properties of the resulting GP model; this is the task which an ABCD system (Section 2) is meant to perform in place of human analysts. The capabilities of the GPy Python library cover considerably more features than an ABCD back-end requires since within it all aspects of GP fitting are developer-configurable.

Appendix B. Details & Comparison

The following sub-sections mirror the structure of Section 3, further exploring implementation details and comparing them with the original ABCD.

B.1 The Model Language

The base kernel expressions are reported below, where σ^2 is variance:

- $WN(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2 \delta_{\boldsymbol{x},\boldsymbol{x'}}$, where $\delta_{\boldsymbol{x},\boldsymbol{x'}}$ is the Kronecker delta function
- $C(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2$
- $LIN(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2(\boldsymbol{x} - \boldsymbol{c})(\boldsymbol{x'} - \boldsymbol{c})$, where $\boldsymbol{c}$ is the horizontal offset (a polynomial root in multiplications)
- $SE(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2 \exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x'})^2}{2l^2}\right)$

- $PER(\boldsymbol{x}, \boldsymbol{x'}) = \sigma^2 \frac{\exp\left(\frac{\cos\left(\frac{2\pi(\boldsymbol{x}-\boldsymbol{x'})}{p}\right)}{l^2}\right) - I_0\left(\frac{1}{l^2}\right)}{\exp\left(\frac{1}{l^2}\right) - I_0\left(\frac{1}{l^2}\right)}$, where p is the period and I_0 is the modified Bessel function of the first kind of order zero

The sigmoidal kernels used by the change operators are constructed as follows:

- Choosing a sigmoidal function sig (the used one is $\frac{x}{1+|x|}$)
- Can define a step-function (and its reverse) by scaling sig to the range $[0, 1]$: $\sigma(\boldsymbol{x}) = \frac{1}{2}\left(1 + sig(\frac{\boldsymbol{x}-\boldsymbol{l}}{s})\right)$ and $\overline{\sigma}(\boldsymbol{x}) = 1 - \sigma(\boldsymbol{x}) = \sigma(\boldsymbol{x}; s \to -s)$, where l and s are location and slope parameters
- Can then also define sigmoidal versions of indicator functions: $hat(\boldsymbol{x}) = \frac{\sigma(\boldsymbol{x})+\overline{\sigma}(\boldsymbol{x}+\boldsymbol{w})-1}{h(\boldsymbol{w})}$ and $well(\boldsymbol{x}) = \frac{2-\sigma(\boldsymbol{x})-\overline{\sigma}(\boldsymbol{x}+\boldsymbol{w})}{h(\boldsymbol{w})}$, where $\boldsymbol{w}$ is the (strictly positive) window width and $h(\boldsymbol{w})$ is the maximum numerator value (i.e. the hat height or well depth), scaling the expressions to 1

The sigmoidal kernels are then straightforward to construct:

- Sigmoidal (S) and Reverse Sigmoidal (Sr) kernels:
  $S(\boldsymbol{x}, \boldsymbol{x'}) = \sigma(\boldsymbol{x})\sigma(\boldsymbol{x'})$ and $Sr(\boldsymbol{x}, \boldsymbol{x'}) = \overline{\sigma}(\boldsymbol{x})\overline{\sigma}(\boldsymbol{x'})$
- Sigmoidal Indicator (SI) and Reverse Sigmoidal Indicator (SIr) kernels:
  $SI(\boldsymbol{x}, \boldsymbol{x'}) = hat(\boldsymbol{x})hat(\boldsymbol{x'})$ and $SIr(\boldsymbol{x}, \boldsymbol{x'}) = well(\boldsymbol{x})well(\boldsymbol{x'})$

Only WN, C and SE are already present in GPy, while the rest had to be newly implemented; more specifically, GPy does provide versions of LIN and PER, and both were tested but eventually discarded in favour of re-implementations of the original ABCD's. GPy's LIN kernel is simpler than ABCD's in that it does not include the offset value $\boldsymbol{c}$, meaning that any covariance which is truly linear but NOT through the origin would require the sum kernel of $LIN + C$. This would be reasonable from a purity-of-model point of view, making the presence in covariance of a vertical shift immediately visible in symbolic form (vs seeing $c \neq 0$), but in practice it is needlessly cumbersome, both computationally and combinatorially, when it comes to polynomial kernels, e.g. the kernel space exploration depth of, say, $LIN \times LIN \times LIN$ is shallower than $(LIN + C) \times (LIN + C) \times (LIN + C)$ (though an ad-hoc production rule could shorten it at the price of making rounds more expensive). Secondly, not including the intrinsic c parameter removes the useful side-effect of being able to simply read out polynomial roots in products of LINs, which is an important feature on its own especially in describing fit kernels with text. GPy's PER kernel is in fact the one on which ABCD's is based, which suffers from the opposite problem of the above LIN in that it does allow vertical shifts while modelling periodicity, which, given the considerably higher complexity than a simple line, makes it more competitive than it needs to be in fitting data which is not periodic (but has, say, two similar peaks). Specifically, ABCD's PER is the purely-periodic component of GPy's (MacKay's standard periodic kernel) (Duv, 2014); given the direct availability of the latter, both were tested and the utility of the former was re-verified: ABCD's PER on its own is able to model, say, $cos(x)$, but not, say, $cos(x) + 1$ (instead requiring the addition of C). While the GPy-ABCD uses the ABCD versions of the above two kernels, it does not share the same sigmoidal ones. The original ABCD SI and

SIr formulae were the same as the S and Sr ones but with the product of opposite-slope sigmoidals in place of every sigmoidal, i.e. $SI(\boldsymbol{x}, \boldsymbol{x'}) = (\sigma(\boldsymbol{x})(1 - \sigma(\boldsymbol{x})))(\sigma(\boldsymbol{x'})(1 - \sigma(\boldsymbol{x'})))$ and $SIr(\boldsymbol{x}, \boldsymbol{x'}) = (1 - \sigma(\boldsymbol{x})(1 - \sigma(\boldsymbol{x})))(1 - \sigma(\boldsymbol{x'})(1 - \sigma(\boldsymbol{x'})))$; there are a few intertwined merits to the versions GPy-ABCD uses:

- the height/depth is fixed to 1 by the denominator and therefore it does not indirectly depend on $(\boldsymbol{x} - \boldsymbol{x'})$ (which would require further scaling by fitting parameters)
- computing the gradients is less computationally intensive
- the window start and end locations are more distinctly identifiable

Being the most complex in the grammar, the CW kernel is unsurprisingly the one requiring the most care in implementation in order to reduce numerical instability and increase result consistency; to this end, many variations of the SI kernel were tried, in both mathematical nature of *sig* and specific parametrisations. In the first respect the tried *sig*s were, in order of decreasing computational complexity, $tanh(x)$, $\frac{x}{\sqrt{1+x^2}}$ and $\frac{x}{1+|x|}$, where the last (and ultimately selected) one has the peculiarity of a very steep derivative culminating in a sharp (removable) discontinuity. In the second respect, some tried parametrisations were start and end locations, central location and width, and start location and width, with the last being ultimately selected; to the same end, in the currently running version the slope parameter is fixed to a constant, thus reducing the kernel complexity and making it more competitive against those with fewer parameters. An issue has been opened on GPy's repository regarding possibilities of alternated fitting and relative constraining of specific parameters (the two implicit window locations), which would aid convergence in this kernel's fitting machinery. Implementation wise, GPy's `BasisFuncKernel` was used as a base for all sigmoidal kernels, allowing the definition of kernel and gradients by their components, i.e. by σ, $\overline{\sigma}$, *hat* and *well*, then letting GPy combine them on its own.

B.2 Kernel Expressions & Simplification

One definition is required before listing the simplification rules:

Definition 2 *A kernel function is* ***stationary*** *if it has no dependence on* $\boldsymbol{x}$ *and* $\boldsymbol{x'}$ *except through* $\boldsymbol{x} - \boldsymbol{x'}$*, meaning that it is not affected by equal shifts in both points*

All base kernels in use here are therefore stationary except for LIN and the sigmoidal ones. The simplification rules are the following (addition and multiplication commutativity, associativity and distributivity apply):

- WN is addition-idempotent, multiplication-idempotent and also acts as multiplicative-zero for stationary kernels, therefore there can be at most one per sum and one with no other stationary factors per product
- C is also addition-idempotent and multiplication-idempotent, but it acts as multiplicative-one, therefore there can be at most one per sum and it is in products only when alone
- SE is multiplication-idempotent
- Since all base kernels include a variance parameter, when in a product they can all be removed in favour of a single product-wide variance

Code-wise GPy-ABCD's design is very different from the original ABCD, having had the benefit of hindsight to globally generalise ad-hoc structures and procedures within a broadly similar algorithm, but there is an important difference in functionality when it comes to expression simplification. The original ABCD has two global operating modes when it comes to expression handling (ABCD-Interpretability and ABCD-Accuracy), the key difference being that the former only works with sum-of-products-form kernel expressions (i.e. it immediately simplifies any nested form even during model space exploration), while the latter completely avoids this type of simplification, returning nested expressions as outputs too, which are not describable in text. Each mode is named after its advantage, the idea being that forcing the exploration to go through specific expression forms achieves eventual model explainability at the price of a slight bias on the process. GPy-ABCD avoids this choice by taking the best of both paths: it does not enforce canonical form during model space search (still simplifying as needed) and it only reduces to sum-of-product form at the very end to add explainability (the difficulty here is only the aforementioned care in handling parameters during post-fit rearranging).

B.3 The Model-Space Search

The main model-space search function exported by the package is:

```
explore_model_space(X, Y,
    start_kernels, p_rules, utility_function, rounds, buffer, dynamic_buffer,
    verbose, restarts, model_list_fitter, optimiser)
```

where only `X` and `Y` are required since the rest have default values. The following is the model-space search algorithm it implements:

- Every GP regression of `Y` by `X` performs a set number (`restarts`) of random initial-parameter-values restarts in order to increase confidence in having converged to global minima/maxima, and its model score is given by the provided `utility_function`
- The search begins by fitting the provided `start_kernels`, acting as a 0^{th} round, then a set number (`rounds`) of standard rounds is executed, iteratively generating and fitting candidate kernels
- In each round, a specific number (`buffer`) of the best-scoring kernels which have not yet been round-inputs is selected for further processing (not just from the previous round, making `buffer` the beam width in the context of beam search)
- Every input-kernel is "expanded", meaning that the provided production rules (`p_rules`, see Section B.3 for examples) are applied to it, generating new expressions which are then filtered for newly-created and previously-encountered duplicates; the remaining ones are fit
- Through the `model_list_fitter` and `optimiser` arguments the user has the options of customising, respectively, how to parallelise the fitting of a list of kernels, and which of the fit optimisers available in GPy to use
- If `dynamic_buffer` is true, then the number of input kernels is higher in earlier rounds (when expressions are simpler) and lower in later ones (in the context of beam search this is just progressively narrowing the beam). This is useful to avoid premature focus on particular expressions when the data shape is particularly complex, giving

a broader spectrum of simple models a chance; in simpler cases this instead reduces overall computation time since less time is spent on fitting needlessly complex models
- Intermediate results are printed during exploration if `verbose` is true (but for the moment the search cannot be interrupted and has to complete the predetermined number of rounds before returning)
- The returned values are:
  - `sorted_models`: all fit models ordered by decreasing fit score
  - `tested_models`: a list of lists of fit models, one per round
  - `tested_k_exprs`: the list of all fit kernel expressions
  - `expanded`: the list of all fit models which have been expanded in a round
  - `not_expanded`: the complement of `expanded` with respect to `sorted_models`

GPy-ABCD provides some ready-made lists of starting kernels and production rules, but, more importantly, it also provides the tools to write custom ones; the default ones are:

Start kernels all base kernels except for SE (since as a 1^{st} round seed it is much too adaptable, obscuring more specific initial ones) plus 2^{nd} and 3^{rd} order polynomials, a vertically shifted PER and both change-type kernels with LIN as an argument (since they are sufficiently simple cases of each for the purpose of capturing the change pattern early if present). That is: WN, C, LIN, PER, $LIN \times LIN$, $LIN \times LIN \times LIN$, $PER + C$, $CP(LIN, LIN)$, and $CW(LIN, LIN)$

Production rules using E to indicate any kernel expression and B to indicate a base kernel, basic rules to span the base model space ($E \to E + B$, $E \to E \times B$, $B \to B$), simple-case change operators to span the whole model space ($E \to CP(E, LIN)$, $E \to CP(LIN, E)$, $E \to CW(E, LIN)$ and $E \to CW(LIN, E)$), simple expression reductions ($E + E2 \to E$ and $E \times E2 \to E$), and a few heuristic shortcuts to commonly reached forms ($E \to E \times (B + C)$, $E \to B$)

The original ABCD is different in starting kernels, production rules and overall algorithm:

- It has no 0^{th} round, and its 1^{st} round is seeded by the result of applying the production rules to the simple WN kernel
- The production rules used are the same as GPy-ABCD's except for the absence of the ones discouraging SE in favour of higher-order curves and replacing the change-kernel ones with pairs of both simpler and more complex versions: $E \to CP(E, C)$, $E \to CW(E, C)$, $E \to CP(C, E)$, $E \to CW(C, E)$, $E \to CP(E, E)$ and $E \to CW(E, E)$
- The overall algorithm is not a beam search but a simple greedy search using exclusively the single best model from the previous round to seed the next one, therefore possibly excluding a better model from, say, two rounds before

B.3.1 Model Selection Method Issue

The choice of utility function with which to score fit models is not free from complications; comparing implementations, the original ABCD uses Bayesian Information Criterion (BIC), while GPy-ABCD allows the user to provide their own arbitrary function and contains a few basic ones (BIC, Akaike Information Criterion (AIC), Akaike Information Criterion corrected for small sample size (AICc) and a Laplace Approximation of Leave-One-Out

Cross-Validation error (LA-LOO) (Vehtari et al., 2016)) but currently also defaults to BIC. Putting aside LA-LOO, which does not meet the requirement of using model complexity to balance closeness of fit, there are however some statistical issues with using the above information criteria, the most important one worth mentioning here being that they all assume that the parametric distribution family under consideration contains the true model (Konishi and Kitagawa, 2007), the opposite of which is precisely the usefulness of an ABCD-like model search. A second issue is that these criteria assume the data is independently drawn from said true distribution, while GPs as a method operate precisely on the assumption of ordered correlation. In practice these criteria do a reasonable job of ordering models by goodness-of-fit, but more statistically sound alternatives (the implementation of which is not straightforward) are worth exploring: Bayesian Predictive Information Criterion (BPIC) (Ando, 2007) and Generalised Information Criterion (GIC) (Konishi and Kitagawa, 2007) explicitly addresses the "true distribution" issue, but other criteria worth examining are Widely Applicable Information Criterion (WAIC) or Widely (Applicable) Bayesian Information Criterion (WBIC) (Watanabe, 2013), and perhaps combinations of model complexity penalties with Leave-One-Out error approximations (more likely by empirical justification rather than statistical argument).

B.4 Model Description

GPy-ABCD's model description procedures cover only the simplest part of the original ABCD implementation's, since the latter produces pages of graphs and parameter analyses individually for each product in the sum-of-products form. This extended type of output does not match GPy-ABCD's purpose, but its post-search nature means it could be developed as a separate module taking in a kernel expression and its GPy counterpart.

B.5 Differences Summary

Most differences stem from the core fact that GPy-ABCD is meant to be an open back-end service rather than a full system unto itself; because of this it is modular and developer-friendly, allowing the configurability and extensibility required to perform model searches of arbitrary complexity and constraints. With respect to the scope of analysis and output type, the original ABCD produces documents of tens of pages which include multiple plots, tables and details on how each identified component affects the full model, while GPy-ABCD is only intended to produce the model itself and a short paragraph describing it. As for the method, unlike the original ABCD's, GPy-ABCD's kernel space search algorithm allows configurable starting conditions and the expansion-candidates selection is able to indirectly backtrack to previous rounds' results; this means that the explored kernel space can be different from the original system's in starting conditions, evolution steps and weights on the directions of expansion.

Appendix C. Evaluation Details

The following sections match the evaluation hypotheses in Section 4

C.1 Synthetic Data

The evaluation procedure with synthetic datasets is straightforward: produce noisy data from functions exhibiting features the system is meant to be able to capture and verify that the top search results contain the base kernel version of the used formula. For example: the "obvious" kernel for noisy data produced from $y = 2x\cos(\frac{x-5}{2})^2$ should identify the interaction of linearity and of vertically-shifted periodicity, i.e. it should be $LIN \times (PER + C)$. Tests were performed for various combinations of up to 3 base kernels both with and without permanent or temporary kernel changes, and the system successfully and consistently identified each combination with one source of difficulty and one exception:

- PER is able to achieve pathologically competitive scores when fitting complex non-periodic data by converging to periods considerably shorter than any significant data curvature, producing fits which are obviously wrong to a human (a behaviour shared by GPy's own non-purely-periodic kernel version)
- CW kernels with non-stationary outer parts (e.g. $CW(LIN, \cdots)$; see Section B.2) are almost never identified; this makes sense since the vertical shift induced by the window portion is highly unlikely to match the one required by the non-stationary outer kernel (e.g. a straight line having to stop and then start again exactly at the required height after the window)

Usability-wise the former point is less of an issue for a human user (who can choose any of the top-scoring models based on their own criteria) than it is for a broader framework automating the use of GPy-ABCD; in this case reasonable options are adding a simple closeness-of-fit score filter on the top-scoring models, or one which checks for explicitly periodic features to compare to the identified periods (e.g. a Fourier transform). Addressing the second point is not straightforward since this behaviour is technically correct. Since two nested CP kernels can fit these scenarios at the cost of more parameters, two possibilities come to mind: either this combination could be encouraged by additional production rules, or a new version of CW with an additional vertical shift could be implemented (i.e. a C added to the post-window side of the non-stationary kernel); in both cases the identified parameters of said kernel would however not be reliable (e.g. the roots of a polynomial which is vertically shifted on the right side of the window).

C.2 Original ABCD Data

Regarding the 2$^{\text{nd}}$ evaluation point, "similar" system behaviour necessarily has to be reduced to similarity in final fit KE given the large design differences (see Section B.5). Some datasets with corresponding original-ABCD analyses are publicly available, and, taking the above into account, they can be used to evaluate GPy-ABCD's output in the above respect, but unfortunately not in explored kernel space details or in computational efficiency of single fits (not necessarily useful in any case given the different technology stack). Because the design differences are a core part of what is being evaluated, although it would be possible to make GPy-ABCD run an algorithm which is very close to the original (identical except for having to prioritise models from previous rounds if better than those strictly in the last one), GPy-ABCD was run on default search parameters with only minor tweaks to compensate for the systems' different intended use cases: GPy-ABCD default settings are

for an initially-broad and then narrowing few-round search on reasonably small datasets (100 points or so) whose complexity is not exceptional (the typical data from FRANK's queries is small in size and "shallow" in terms of expression "depth"). For each of the available analyses, the original ABCD's (single-expansion, naive-greedy) model search was run for 10 rounds and with 10 random parameter restarts for each fit model (Lloyd, James Robert et al., 2014). Based on this, GPy-ABCD's search parameters were set to the same number of restarts and an equivalent number of expansions (though not necessarily of maximum "depth"): 5 rounds with a (non-dynamic) buffer of 2. Given the known dataset complexity and to allow fully comparable expressions between systems, the final tweak from GPy-ABCD's default parameters was to re-enable the SE kernel (normally excluded since it tends to dominate early rounds while being the least interpretable). Comparing results, no noticeable differences in closeness of fit were present, and GPy-ABCD's expressions were generally simpler than the original's. This is expected since 5 2-buffer beam-search rounds are unlikely to be as deep as 10 naive-greedy ones, but a point of note is the frequent use of change-kernels in the original ABCD's, while GPy-ABCD seems to avoid them. At this point it is not clear why this should be the case: assuming equally effective change-kernel fits in both systems (the synthetic data evaluation does not suggest inadequacies on this front), and given the equal number of added parameters (change locations), the complexity penalty in model score (BIC in both systems) should have the same effect in steering the model-space search focus. It is not inconceivable that the base sigmoidal function difference ($\frac{x}{1+|x|}$ for GPy-ABCD and *tanh* for the original) could play a part in this, though the precise mechanics are not obvious; differences in fit effectiveness between the two underlying frameworks (Python & GPy vs Matlab & GPML) are an unquantified factor.

C.3 Effectiveness in Original Purpose

Feature-wise GPy-ABCD is capable of fulfilling everything it was intended for within FRANK, i.e. handling inputs and outputs of requests to fit data of extremely varied shapes, including producing text descriptions, which allows the addition of functional-form-based queries to FRANK's arsenal. However, as the paper so far will have made clear, this type of modelling is not without its downsides, particularly when in the context of QA, the principal issue being the high computational expensiveness of the full model-space search (since even a single GP regression with the required multiple random parameter restarts takes some time). GPy-ABCD's configurability-oriented design is in part due to this issue, allowing FRANK to run it in full only for datasets of a few hundred points, beyond which it could be run on increasingly more constraining parametrisation profiles (e.g. fewer rounds, production rules or base kernels); alternatively, FRANK could just feed it smaller uniform samples of the datasets to return approximate solutions quickly, and then leave the user the choice of whether to proceed to more expensive models. This level of consideration and weighting of alternatives for the set of available modelling methods (which can overlap over different query types) would be resolved precisely by considerations such as the above, always taking care to inform the user of the reasons and possibly applied restrictions (and giving the option to enforce a choice regardless); that is to say that FRANK having to work around this computational expensiveness by offering the user more choices is an acceptable outcome, and one which plays well with other competing modelling components.