Flame: An Open Source Framework for Model Development, Hosting, and Usage in Production Environments

Manuel Pastor (manuel.pastor@upf.edu)
Universitat Pompeu Fabra https://orcid.org/0000-0001-8850-1341
José Carlos Gómez-Tamayo
Universitat Pompeu Fabra https://orcid.org/0000-0003-4709-6030
Ferran Sanz
Universitat Pompeu Fabra https://orcid.org/0000-0002-7534-7661

Software

Keywords: Modeling framework, modelling tools, reproducibility, model management, workflow, QSAR, model integration, web-interfaces, in-silico toxicology

Posted Date: November 18th, 2020

DOI: https://doi.org/10.21203/rs.3.rs-107430/v1

Version of Record: A version of this preprint was published at Journal of Cheminformatics on April 19th, 2021. See the published version at https://doi.org/10.1186/s13321-021-00509-z.

Abstract

This article describes Flame, an open source software for building predictive models and supporting their use in production environments. Flame is a web application, with a Python backend and a web-based graphic interface, which can be used as a desktop application or installed in a server receiving requests from multiple users. Models can be built starting from any collection of biologically annotated chemical structures, since the software supports structural normalization, molecular descriptor generation and machine learning building, using predefined workflows. The model building workflow can be customized from the graphic interface, selecting the type of normalization, molecular descriptors, and machine learning algorithm to be used from a panel of state-of-the-art methods implemented natively. Moreover, Flame implements a mechanism allowing to extend its source code adding unlimited model customization. Models generated with Flame can be easily exported facilitating collaborative model development. All models are stored in a persistent model repository supporting model versioning. Models are identified by unique model IDs and include detailed documentation formatted using widely accepted standards. The current version is the result of nearly three years of development in collaboration with users from pharmaceutical industry within the IMI eTRANSAFE project, which aims, among other objectives, to develop high quality predictive models based on shared legacy data for assessing the safety of drug candidates.

Introduction

In the last years biomedical data is becoming widely available, thanks to the creation of well-organized repositories like PubChem [1], ChEMBL [2], eTOX [3] and initiatives like FAIR [4] facilitating the access of existing data to the scientific community.

An interesting way of exploiting this vast amount of data is the development of mathematical models connecting the chemical structure of the substances with their biological properties. Such models are not new, Quantitative Structure-Activity relationship (QSAR) were first described in the 60s [5]. QSAR models use regression methods to identify the structural properties linked to quantitative biological properties or directly to predict these properties for new substances. For biological properties characterized using qualitative descriptions (e.g. positive or negative) conceptually similar approaches can be applied using classifiers. The first QSAR models were developed using small series of congeneric compounds, often synthetized and tested *ad-hoc* for the study. Nowadays, large series of structurally diverse compounds can be easily obtained from public repositories. Pharmaceutical companies can also extract these series from their own internal repositories and use them isolated or combined with compounds from external sources. This fact, combined with recent developments in machine learning (ML) and deep learning (DL) methodologies [6] as well as with the implementation of many of these methods in open source libraries [7], create an ideal scenario for the development of predictive models with biomedical application.

Table 1. Examples of ML/DL applications in Biomedical research

ML application	Brief explanation	Reference
Drug discovery	Identification of new bioactive compound.	[6–8]
Toxicity prediction	Identify hazardous substances.	[9–11]
Precision medicine	Personalize medical treatment to patient idiosyncrasy.	[12]
Imaging diagnostics	Identification of abnormalities from imaging.	[13, 14]

Indeed, the use of ML and DL is becoming very popular in biomedical research. A few remarkable models developed recently have been listed in Table 1, as examples of applications of this methodology, illustrating their usefulness.

More and more, the models obtained by the application of ML are seen as valuable business assets. Accurate and appropriately shared models can bring a number of benefits if we are able to make effective use of existing expertise [15]. However, the true capability of a model for solving real world problems critically depends on aspects which are often ignored, mostly related to model implementation, as follow.

Reproducibility

Models must produce the same results when used at different sites or times. This simple, basic requirement, is difficult to meet if (i) the training data is not available and distinctively identified or (ii) the algorithms used are not documented with enough detail or if it is not possible to use exactly the same software (same version, same platform, etc.). The fast evolution of computational tools (both hardware and software) makes challenging preserving a model for some time.

Accessibility

Models are digital assets to which the FAIR accessibility principle can also be applied [4, 16]. Ideally, access to existing models should be facilitated, particularly for models developed in academic environments. In practice, there are barriers related with the intellectual property of the tools required to generate the predictions. This can apply to commercial applications used to generate 3D structures or molecular descriptors, or even the modeling software itself. For this reason, the use of open source alternatives should be prioritized.

Not all accessibility barriers are related to intellectual property issues, and models should be implemented in a way that allow their use in different operative systems (e.g. Windows, Linux, iOS) and platforms (e.g. implemented as a desktop application or as REST [17] web services in centralized servers). This is particularly true in corporative environments, where company restrictive policies about OS or platforms could hinder the access to useful models. Also, and not less important, is to facilitate the model use for non-experts, by providing a friendly end-user interface.

Model management

Models must be labelled with unique identifiers and stored appropriately. This facilitates common tasks like knowing which model was used to generate some prediction or retrieving a certain model cited in a report. This task is hampered by the fact that models are not static entities. Models evolve as the software they use is updated or as the training series are enriched with new compounds for covering a wider chemical space. Consequently, models often have many versions which must be properly identified and stored as well, recording all the changes in the training series and the modeling software.

A separate task is to document the models. Models can be documented with different levels of detail for different purposes. As a minimum, every model must be accompanied by documents allowing to reproduce the algorithm in full, and to understand and interpret the prediction results. Other purposes, like demonstrating to regulatory bodies the quality of the prediction for replacing experimental tests [18], are only seldom required. Therefore, we recommend a layered documentation structure, including basic mandatory information and more detailed optional layers.

Reporting

For the model developer the meaning of a model prediction results is obvious; the model estimates the biological annotations present in the training series. However, users not involved in the process of model building lack this context. This often creates confusion and difficulties for users to interpret the model's prediction, particularly in case of a numerical outcome. For this reason, as a minimum, models results must include explicitly the units in which they are expressed, a brief, concise explanation of how these results must be interpreted, and the level of confidence within which the prediction values must be apprehended.

Every prediction has a certain uncertainty associated as a consequence of the errors present in the training series annotations as well as the limitations of the model predictivity. For this reason, prediction results must be accompanied by a quantitative estimation of the individual prediction error. This estimation cannot be generic, based solely on the error observed for the compounds in the training series and must consider how far the query compound is away from the model applicability domain.

The present article describes Flame, a new modeling framework developed with the aim of overcoming the issues described above and facilitating the development, hosting and use of predictive models in production environments.

Flame was developed in the context of project eTRANSAFE (IMI2 Joint Undertaking under Grant Agreement No. 777365.) which is producing integrative data infrastructures and innovative computational methods to improve the feasibility and reliability of translational safety assessment during the drug development process. For this reason, Flame was originally designed to host predictive models for drug safety endpoints, even if it can be used with other applications in biomedical research.

Implementation

The Flame architecture is illustrated in Figure 1. It consists of a Python library (Flame backend) which can be used from a terminal with command line interface, called from a Jupyter notebook [19] or scripts written shell languages (bash, bat, etc.). It also implements a web server (written in Django [20]) offering the library features as REST services [17] and a complete web interface (written in Angular [21]) providing a rich graphic user interface (GUI).

The GUI can be executed locally as a desktop application, starting the web server in the same computer running the Flame backend. It is also possible to run the Flame backend in a server and access the REST services from a remote client thus allowing to run Flame as a departmental or global prediction service in corporate environments.

Table 2. Main Python classes used in Flame.

Type	Class	Functionality	Input	Output
High-level	build	Generates a model	Training series	Model
	predict	Uses an existing model to generate a prediction for a given input compound	Query compound	Prediction
	manage	Handles (create, delete, export, import, etc.) models in the repository	-	-
Low-level	idata	Processes chemical structures to obtain molecular descriptors as an X matrix and annotations as Y matrix (when provided)	Substance	X (Y) numerical matrices
	learn	Generates a model from X and Y numerical matrices	X and Y numerical matrices	Model
	apply	Uses an existing model to generate a prediction from a X matrix	X numerical matrix	Prediction
	odata	Formats results as human readable output or formats suitable for the GUI	Results	Human or GUI readable output

The Flame backend and the optional flame web server make use of Conda [22] to guarantee a consistent environment, free of library incompatibilities, which can be executed in Linux, Windows, and iOS operative systems.

The code was written using Object Oriented Programming (OOP) as a Python library. The main classes (see Table 2 and Figure 2) can be classified as low-level or high-level. The first carry out simple tasks while the latter make calls to low-level classes to execute model building and model prediction workflows.

For example, the default model building workflow implemented in high-level class *build* (Figure 2) starts from a training series of annotated chemical structures and uses class *idata* to import their chemical structures, normalize them and generate molecular descriptors which are stored in a numerical matrix. This information, together with the compounds biological annotations is passed to the class *learn*, which normalizes the numerical values and builds models using machine learning (ML) tools like Random Forest (RF). This model is stored in a machine-readable format (so called *estimator*, in agreement with the terminology used by the scikit-learn library [7]) suitable to predict the properties of novel compounds. Finally, the class *odata* is used to format the results and produce suitable output. The default prediction workflow (Figure 2) uses exactly the same low level *idata* class to import and pre-process the structures. This workflow design has the advantage of guaranteeing that a prediction uses exactly the same code that was used for model building for equivalent tasks and to produce consistent results. Then, the low-level class *apply* retrieves the *estimator* saved previously during the model building process for generating the prediction results and a call to the *odata* class generates the output.

Most of the building and prediction workflow steps are configurable. For example, we can select the structure normalization algorithms or the molecular descriptors method that will be used. Also, the methods themselves can be configured by adjusting their internal parameters. In Flame, the methods used to build a model and their configurable parameters are defined in a single parameter file (parameters.yml) which can be seen as the model “blueprint”. This file, together with the original training series and the *estimator* generated by *build* is stored in a folder of the model repository. This folder is often called here “the model” and can be considered a complete and comprehensive definition of how a model has been built. Models are used by Flame to predict the properties of new compounds using the *predict* workflow (see Figure 2) and can be saved, compressed, backed-up or transmitted between Flame instances installed in different computers. In any of these cases, Flame guarantees that the predictions are reproducible. In this sense, Flame models can be seen as self-contained prediction engines. Flame provides commands to export and import models as a single binary file, consisting in the compressed version of the model folder. On import, the version of the software used to generate the model is checked to guarantee full compatibility and reproducibility.

The use of the parameter file described above offers only a limited customization, since the user can select only among the algorithms and methods implemented natively in Flame. To overcome this limitation the model workflows do not call the low-level classes directly, but uses a child class instead, inheriting all the parent class properties, which is stored locally within the model folder (see Figure 3). For simple models, this is transparent to the user and is the exact equivalent to calling the Flame classes directly. However, advanced uses can override any method of the locally stored child class. This strategy allows unlimited model customization. For example, users can make use of external tools for generating molecular descriptors, include extra steps in the model building or prediction workflow, adapt the output to generate customized reports, etc. Since these changes are written in the local instance of the code (located within the model folder) they do not affect other models. Moreover, these changes are preserved when the model is saved or exported.

Results

Model building features

Flame can build predictive models starting from a single file in SDFile format containing the structures and the biological properties of the training series. The default model building workflow takes care of reading the structures, normalizing them, generating molecular descriptors, scaling their values and building a machine-learning model which is saved in a format suitable for predicting the properties of new compounds.

Flame provides defaults for methods and parameters but the user can customize them, either editing the parameter file parameters.yml when using Flame in command line mode or using the model building dialogue (Figure 4), when using the Flame GUI.

Table 3 describes the methods implemented natively in Flame. All of them make use of open source libraries. The choice of models can be easily extended to include commercial products or external tools, using the code overriding technique described at the implementation section.

Table 3. Overview of the main modeling methods and tools implemented natively in Flame.

Modeling task	Method	Source
Structure normalization	Standardiser	[23]
	ChEMBL pipeline	[24, 25]
Molecular descriptors calculation	RDKit properties	[26]
	RDKit md	[26]
	RDKit Morgan fingerprints	[26]
Scaling	Raw	-
	Autoscaling	[7]
Machine learning	RF	[7, 27]
	SVM	[7, 28]
	PLS	[7, 29]
	XGBOOST	[30]
	Conformal regression	[31, 32]

Typically, models are built starting from a collection of annotated chemical structures, but Flame can also use as input a tab separated (TSV) table with pre-calculated molecular descriptors and annotations.

Another option, rarely found in other modeling frameworks (but present in OCHEM [33]), is the possibility to use as input the prediction results of other models present in the repository. This option, called in Flame “model ensemble” is interesting for combining the results of multiple qualitative models into a single result representing the majority voting. The prediction results of an ensemble of quantitative models can also be combined using the individual predictions mean or median. Regressors and classifiers can also be applied to the model ensemble output to generate a smarter result combination and obtain better predictions. When the ensemble models provide an estimation of the individual prediction error this information is considered, using appropriate probabilistic methods, to generate an estimation of the final prediction error. The description of these algorithms is beyond the scope of the present work and will be published in a separate article.

The last step of model building workflows is an estimation of the model’s quality using cross-validation which presents the user information about the model’s goodness of fit, the model predictive quality and some characteristics of the training series (e.g. value distribution). Since Flame can use diverse ML methods, we tried to generate comparable output for facilitating the selection of the best methods and parameters. The values shown are summarized in Table 4, for qualitative and quantitative endpoints.

Table 4. Model quality parameters shown in the Flame GUI.

Endpoint type	Parameter	Definition
Qualitative	Sensitivity (fitting and prediction)	$\frac{TP}{TP+FN}$
	Specificity (fitting and prediction)	$\frac{TN}{TN+FP}$
	MCC (fitting and prediction)	$\frac{(TP*TN)-(FP*FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$
Quantitative	SDEC (fitting)	$\sqrt{\frac{\sum(Yexp-Ypred)^2}{n}}$
	SDEP (prediction)	$\sqrt{\frac{\sum(Yexp-Ypred)^2}{n}}$
	r^2 (fitting)	$\frac{\sum(Yexp-Ypred)^2}{\sum\sqrt{(Ymean-Ypred)}}$
	q^2 (prediction)	$\frac{\sum(Yexp-Ypred)^2}{\sum\sqrt{(Ymean-Ypred)}}$
Conformal models	Conformal Coverage	$\frac{Samples\ inside\ confidence\ boundaries}{Total\ number\ of\ samples}$
	Conformal Accuracy	$\frac{Samples\ predicted\ correctly}{Total\ number\ of\ samples}$
	Mean interval (only quantitative)	$\frac{\sum\lvert Ymax-Ymin\rvert}{n}$

The Flame GUI provides additional information, oriented to diagnose the quality of the model and the training series, as shown in Figure 5. For qualitative endpoints (left side of Figure 5), the confusion matrix is shown as a 2x2 matrix and as a “radar plot” expressing in the radius of the sections the relative size of true positive, true negative, false positive and false negative results. This information is shown separately for the model fitting and prediction, the latter being calculated using cross-validation methods selected by the user (default to five k-fold). In addition, Flame displays a scatterplot of the training series, obtained by running a Principal Component Analysis (PCA) with the calculated molecular descriptors and showing the

2 first Principal Components (PCs). Objects (compounds) are colored red or blue according to their biological annotations (respectively, positive or negative). The positive and negative ratio of substances in the training series is depicted using a pie chart.

For quantitative endpoints (right side of Figure 5), apart from the parameters mentioned in table 4, the interface shows scatterplots of fitted/predicted values versus the experimental annotations. For conformal models, the confidence interval for the defined confidence level is also shown. In a separate tab, Flame displays a scatterplot of the training series, like the one shown for qualitative endpoints, but in this case the substances are colored using the continuous scale included in the plot. The distribution of the annotation values is shown using a violin-type plot, which offers valuable information to diagnose the presence of a skewed value distribution or the presence of outliers. All the graphics representing the training series are interactive, and hovering the mouse cursor over the dots allows to display the 2D structure of the compounds they represent.

The model quality reports described above are persistent. All this information is stored within the model folder and can be retrieved and shown for every model in the repository at a later time.

Error handling

A very important feature that should be implemented in any modeling software aiming to solve real-life problems, is error handling. A workflow can fail for many reasons: molecules can have a wrong structure, contain metals or water, the model building can also fail when the annotations are not correct. For this reason, a lot of effort was devoted in Flame to implement appropriate error handling methods, removing molecules that cannot be processed and producing suitable output, which is shown in the command line or the GUI. Modelers know that the number of potential sources of error is high and Flame cannot claim to be able to handle all error types. However, years of development and use by different modeling teams, established Flame as a rather robust software.

Model predictions

Any models stored in the repository can be used to predict the properties of a new substance simply entering a SDFile with the structure of the compound(s) to predict. The prediction workflow will then apply to this file the same pretreatment, molecular descriptors calculation and x-matrix scaling used for the training series, using exactly the same source code, thus guaranteeing the maximum consistency of the results. The molecular descriptors obtained are then projected using the stored estimator. The prediction results can be qualitative or quantitative, depending on the nature of the training series annotations. Models built using conformal regression [32] generate additional information about the prediction uncertainty; for quantitative endpoints they provide a confidence interval, while for qualitative (binary) endpoints the result of the prediction can be "uncertain", meaning that the model cannot ascertain if the result is positive or negative. In both cases, the uncertainty is reported at a given probability (confidence level of CI or probability that the result is correct, respectively) which must be defined by the user.

Models are watermarked during the building process and have a unique ID associated. This ID is read when the model is used for prediction. This means that predictions keep record of the model version used to generate it, guaranteeing a full traceability.

As stated in the introduction, prediction results are often difficult to understand and interpret by users not involved in the model building. For this reason, the Flame GUI presents the prediction results in different formats, decorated with extra information aiming to facilitate the result interpretation and its use for decision making.

As shown in Figure 6, results are displayed in three alternative views. First, they are presented as a list, including for every predicted compound its name, 2D structure and prediction result, as well as uncertainty information when available. This list is paged, searchable and can be reordered. It can also be exported to Excel or PDF formats, printed, or copied to the clipboard. Clicking in any of the list items displays a more detailed report for a single compound, showing again the compound name, structure and prediction result but also information about how to interpret the result (extracted from the model documentation), and a list of the closest compounds in the training series with their biological annotations. The similarity is computed using the same molecular descriptors used for building the model.

When the model used for the prediction is an ensemble, the prediction report shows the individual result of the low-level models and combined result (Figure 7). For conformal binary classifiers (left of Figure 7) the graphic shows the low-level model prediction results, indicating if the query compound belongs to class 0 (negative), class 1 (positive), both of them (inconclusive type I) or neither (inconclusive type II). For conformal quantitative models (right of Figure 7), the predictions are shown with the corresponding confidence intervals.

Finally, the prediction results are also represented projected on a scatterplot of the training series PCA scores, generated as explained in the previous section (Figure 5). The aim of this representation is to show whether the predicted compound belongs to a region of the chemical space well represented by the training series or if it falls in a desert region. In this representation, the training series compounds can be displayed as grey dots or colored by the biological annotation. Predicted compound can be displayed as green circles with the compound names, as red dots or as dots colored by the compound distance to model (DModX, see [34]). A high DModX value indicates that the predicted compound has “original features” not present in the training series which can be detrimental for the prediction quality.

Finally, it should be mentioned that the predictions are stored in a persistent prediction repository and therefore it is possible to revisit previous predictions, until they are actively removed by the user.

Model management

Once a model is built, it is stored in a separate folder of the model repository. This folder can contain multiple versions of the model. As a minimum, there is a *dev* version which must be considered as a

“sandbox” used only for model development, and which is overwritten every time the model is re-built. Precisely for this reason, the *dev* version cannot be used for prediction. Model versions which the model developer considers worth storing should be ‘published’ to generate version 1, 2, etc.

The main GUI window shows a list (Figure 8) where models can be browsed and selected. Every model is identified by a name and version and labelled by Maturity, Type, Subtype, Endpoint and Species. The labels are defined by the end-user and can be used to filter the models shown, thus making it easier to find models for a certain endpoint, species, organ, etc.

Both the command mode interface and the GUI provide model management commands for creating new models, publishing a model version, deleting a whole model tree with all the versions or any single model version, etc.

Models can be exported using a command that produces a compressed version of the whole model folder. This file can be easily stored, backed-up, or sent in electronic formats (e.g. as an e-mail attachment). Once imported in any Flame instance, the model is copied to the model repository and becomes fully functional. During the import step, the versions of the software libraries used for generating the models were checked and in case of version mismatches a warning message is issued.

Model documentation

Flame models are documented using a template based on the QMRF [35], reusing our previous experience in model documentation [36]. When the model is built, Flame automatically completes in this template the fields describing the modeling methodology and quality. This half-completed document should be edited by the modeler, using the GUI or editing a documentation file in yaml format using a text editor and re-importing it into the model. In either case, the model documentation is stored at the model folder, and is included when the model is exported or published.

The model documentation has been split in three sections: General Model information, Algorithms and Other information. The first and third sections should be completed by the modeler while Flame automatically completes most of the second section. The additional file ‘*BZR.yaml’* contains an example of human-readable file in yaml format, suitable for being imported into a Flame model, with all the items included in these three sections. The additional file ‘*model documentation GUI.pdf’* contains a PDF files showing how the model documentation is presented to the user in the Flame GUI.

Performance

In a typical modeling workflow, the same code (structure normalization, molecular descriptors calculation) is run for every compound in the input series, both for training series and prediction series. This makes it simple to speed up the computation by splitting the series in *n* sub-series and assigning them to different computation threads, which are run in different CPUs. Flame has an option for running parallel tasks related to the molecular descriptors calculation, obtaining nearly linear speedup. Another time-consuming step is the model building and validation. By default, Flame applies the multitasking

implemented in the ML libraries (e.g. in scikit-learn, it is used for cross-validation or grid-search and in XGBOOST it makes an excellent use of multi-CPU computing power). Use of GPUs is under development and a special Flame version supporting GPUs is planned to be released in the future, facilitating the efficient use of deep learning within the framework.

Additionally, during model development it is a common practice to rebuild the model repeatedly using diverse machine learning settings to optimize them. To speed up this process, Flame stores intermediate results of the calculation (e.g. the molecular descriptors matrix) thus saving the work of re-computing them in every cycle.

Discussion

At the introduction we justified the present work by the existence of unmet needs in the practical use of predictive models. After reporting our proposed solution, we can briefly revisit these needs and discuss if Flame addresses them.

Reproducibility

Models generated and stored in Flame are fully reproducible across Flame instances and can be easily exported and imported, always obtaining the same results. The use of controlled Conda environments, and the tagging of the versions used during the model generation guarantee a full control of the software and software versions used. However, Flame cannot avoid the obsolescence of the software and hardware. For medium to long term model storage, saving images of the whole system using docker or virtual machines is recommended.

Accessibility

Flame is open source and uses only open source software. It is available in the most popular operative systems (Linux, Windows, and iOS) and can be used as a desktop application with a rich GUI, from the command line, integrated in scripts, in Jupyter notebooks or as a web service. The GUI was designed for non-expert users, but experienced modelers can customize the models without limitations. Additionally, Flame installers are offered for Windows and Linux to facilitate installation to non-expert users. These installers are also provided in a portable fashion, which includes all the libraries needed to run Flame without the need of Internet connection. This feature is needed in corporate environments where security is critical and Internet connection is either blocked or filtered

Governance

Flame incorporates advanced model management tools, supporting the whole model development cycle. Models can be developed, improved, and stored in a persistent model repository, where they can be labeled using up to four types of keywords. Models are also thoroughly documented using widely accepted standards and given a unique ID. The documentation is organized in sections using a structure close to the layered approach proposed in the introduction.

Reporting

Flame predictions are presented to the users in a variety of formats, some of them specifically designed to facilitate the interpretation by non-expert users, providing contextual information about the biological annotations and the result interpretation. Whenever the model allows, the prediction result is presented together with information about its uncertainty, using rigorous formalisms (e.g. conformal regression) expressed in formats familiar to experimentalists (confidence intervals).

Conclusions

We presented Flame, an open source modelling framework which can be used for the easy development of QSAR-like models. The incorporated model building workflow only requires the input of a single annotated SDFile to generate a model, using default options. This workflow can be easily customized to use any of the natively supported methods and a variety of method parameters. Moreover, it incorporates mechanisms to implement unlimited customization by using model-linked source code overriding.

Many applications of predictive models depend critically on addressing implementation issues that hinder the use of models in production environments. The modeling framework described in this article solves most of these issues and facilitates a seamless transition from model development to model production with little effort. Models can be easily maintained and stored, as well as exported and imported, facilitating the collaboration between academic and private institutions.

Flame uses innovative methods to combine models by building models based on the results of other models. In the toxicological field, this adds a unique flexibility for combining multiple models addressing the same endpoint, or combining models representing multiple mechanisms contributing to the same endpoint. Some interesting applications of this model combination tool have been obtained and will be published in due time.

Unlike other tools, Flame incorporates a rich web-based GUI, facilitating the model building, administration and use in prediction. Prediction results are presented to the user in different formats, including information like the substances in the training series which are closer to the predicted compound or projections of the query compounds on the training series chemical space.

Flame has been developed in a large European project in which several pharmaceutical companies are testing Flame internally, developing models of interest for them. The feedback obtained in this interaction has been an extremely valuable resource for designing a tool that can help drug developers and drug safety experts in their daily work.

For all these reasons, Flame can be considered a very useful tool with unique features. As yet, it does not incorporate all the modelling tools available, but we plan to keep enriching their features, incorporating other molecular descriptor generators and machine learning toolkits. In this respect our plans are to

expand the Flame user's community beyond the eTRANSAFE consortium and interest developers that can contribute their code in future versions.

Declarations

Availability and requirements

- Project name: Flame
- Project home page: https://github.com/phi-grib/flame (backend), https://github.com/phi-grib/flame_API (web server), https://github.com/phi-grib/flameWeb2 (Angular)
- Operating system(s): Platform independent. Tested in Windows, Linux and iOS
- Programming language: Python, Typescript (Angular)
- Other requirements: Flame uses a Conda environment defining dependencies to other Python libraries
- License: GNU GPL-3.0

Declarations

Availability of data and materials

Flame source code is available at GitHub under GNU GLP-3.0 license at the following repositories: https://github.com/phi-grib/flame (backend), https://github.com/phi-grib/flame_API (web server), https://github.com/phi-grib/flameWeb2 (Angular). No dataset was described nor required to support the conclusions of the manuscript.

Competing interests

The author(s) declare(s) that they have no competing interests

Funding

This work has received funding from the eTRANSAFE project (Grant Agreement No. 777365), developed under the Innovative Medicines Initiative Joint Undertaking (IMI2), resources of which are composed of a financial contribution from the European Union's Seventh Framework Programme (FP7/2007-2013) and EFPIA companies' in kind contributions. The authors of this article are also involved in other related IMI projects which contributed funding, such as TransQST (no. 116030) as well as the H2020 EU-ToxRisk project (no. 681002) and FAIRplus (no. 802750). The Research Programme on Biomedical Informatics (GRIB) is a member of the Spanish National Bioinformatics Institute (INB), funded by ISCIII and FEDER (PT17/0009/0014). The DCEXS is a 'Unidad de Excelencia María de Maeztu', funded by the AEI (CEX2018-000782-M). The GRIB is also supported by the Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR), Generalitat de Catalunya (2017 SGR 00519).

Authors' contribution

MP coded most of the Flame software and prepared the manuscript. JCGT wrote significant sections of the software and contributed to the manuscript. FS contributed to the software concept and design and reviewed the manuscript.

Authors' information

MP is the head of the Pharmacoinformatics laboratory at the GRIB (UPF), and leader of modeling related work packages at eTRANSAFE project. JCGT is a post-doctoral fellow at the MP's Pharmacoinformatics laboratory. FS is the head of the GRIB and eTRANSAFE academic coordinator.

Acknowledgment

We wish to thank Thomas Steger-Hartmann (Bayer AG, Pharmaceuticals, Berlin), Francois Pognan and Nils Oberhauser (Novartis, Basel) for manuscript review and valuable comments.

References

1. Kim S, Thiessen PA, Bolton EE, et al (2016) PubChem substance and compound databases. Nucleic Acids Res 44:D1202–D1213. https://doi.org/10.1093/nar/gkv951
2. Gaulton A, Hersey A, Nowotka ML, et al (2017) The ChEMBL database in 2017. Nucleic Acids Res 45:D945–D954. https://doi.org/10.1093/nar/gkw1074
3. Sanz F, Pognan F, Steger-Hartmann T, Díaz C (2017) Legacy data sharing to improve drug safety assessment: The eTOX project. Nat Rev Drug Discov 16:811–812. https://doi.org/10.1038/nrd.2017.177
4. Wilkinson MD, Dumontier M, Aalbersberg IjJ, et al (2016) The FAIR Guiding Principles for scientific data management and stewardship. Sci Data 3:160018. https://doi.org/10.1038/sdata.2016.18
5. Cherkasov A, Muratov EN, Fourches D, et al (2014) QSAR modeling: Where have you been? Where are you going to? J. Med. Chem.
6. Rifaioglu AS, Atas H, Martin MJ, et al (2019) Recent applications of deep learning and machine intelligence on in silico drug discovery: methods, tools and databases. Brief Bioinform 20:1878–1912. https://doi.org/10.1093/bib/bby061
7. Varoquaux G, Buitinck L, Louppe G, et al (2015) Scikit-learn. GetMobile Mob. Comput. Commun. 19:29–33
8. Vamathevan J, Clark D, Czodrowski P, et al (2019) Applications of machine learning in drug discovery and development. Nat Rev Drug Discov 18:463–477. https://doi.org/10.1038/s41573-019-0024-5
9. Luechtefeld T, Rowlands C, Hartung T (2018) Big-data and machine learning to revamp computational toxicology and its use in risk assessment. Toxicol Res (Camb) 7:. https://doi.org/10.1039/c8tx00051d

10. Luechtefeld T, Marsh D, Rowlands C, Hartung T (2018) Machine learning of toxicological big data enables read-across structure activity relationships (RASAR) outperforming animal test reproducibility. Toxicol Sci. https://doi.org/10.1093/toxsci/kfy152
11. Rabesandratana T (2016) A crystal ball for chemical safety. Science (80-) 351:651. https://doi.org/10.1126/science.351.6274.651
12. Grapov D, Fahrmann J, Wanichthanarak K, Khoomrung S (2018) Rise of Deep Learning for Genomic, Proteomic, and Metabolomic Data Integration in Precision Medicine. OMICS 22:630–636. https://doi.org/10.1089/omi.2018.0097
13. Abraham A, Pedregosa F, Eickenberg M, et al (2014) Machine learning for neuroimaging with scikit-learn. Front Neuroinform 8:14. https://doi.org/10.3389/fninf.2014.00014
14. Shen D, Wu G, Suk H-I (2017) Deep Learning in Medical Image Analysis. Annu Rev Biomed Eng 19:221–248. https://doi.org/10.1146/annurev-bioeng-071516-044442
15. Palczewska A, Fu X, Trundle P, et al (2013) Towards model governance in predictive toxicology. Int J Inf Manage 33:567–582. https://doi.org/10.1016/j.ijinfomgt.2013.02.005
16. Commission E (2018) Turning FAIR Into Reality
17. Roy Thomas Fielding (2000) Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine
18. Rovida C (2020) Internationalization of read-across as a validated new approach method (NAM) for regulatory toxicology. ALTEX 1–29. https://doi.org/10.14573/altex.1912181
19. Kluyver T, Ragan-Kelley B, Pérez F, et al (2016) Jupyter Notebooks – a publishing format for reproducible computational workflows. In: Loizides F, Schmidt B (eds) Positioning and Power in Academic Publishing: Players, Agents and Agendas. pp 87–90
20. Django project. https://www.djangoproject.com/
21. Angular. https://angular.io/
22. Conda. https://docs.conda.io/projects/conda/en/latest/
23. Atkinson F standardizer. https://github.com/flatkinson/standardiser
24. ChEMBL standardiser. https://github.com/chembl/ChEMBL_Structure_Pipeline
25. Bento AP, Hersey A, Félix E, et al (2020) An open source chemical structure curation pipeline using RDKit. J Cheminform 12:1–16. https://doi.org/10.1186/s13321-020-00456-1
26. RDKit: Open-Source Cheminformatics Software. https://www.rdkit.org/
27. Ho TK (1995) Random decision forests. Proc Int Conf Doc Anal Recognition, ICDAR 1:278–282. https://doi.org/10.1109/ICDAR.1995.598994
28. Cortes C, Vapnik V (1995) Support-Vector Networks. Mach Learn 20:273–297
29. Wold S, Johansson E CM (1993) PLS—partial least squares projections to latent structures. In: Kubinyi H (ed) 3D-QSAR in Drug Design, Theory, Methods, and Applications. ESCOM, Leiden, pp 523–550
30. Sharma N (2018) XGBoost. The Extreme Gradient Boosting for Mining Applications. GRIN Verlag

31. Norinder U, Carlsson L, Boyer S, Eklund M (2015) Introducing conformal prediction in predictive modeling for regulatory purposes. A transparent and flexible alternative to applicability domain determination. Regul Toxicol Pharmacol 71:279–284. https://doi.org/10.1016/j.yrtph.2014.12.021
32. Non-conformist. https://github.com/donlnz/nonconformist
33. Sushko I, Novotarskyi S, Körner R, et al (2011) Online chemical modeling environment (OCHEM): web platform for data storage, model development and publishing of chemical information. J Comput Aided Mol Des 25:533–554. https://doi.org/10.1007/s10822-011-9440-2
34. Eriksson L, Andersson PL, Johansson E, Tysklind M (2006) Megavariate analysis of environmental QSAR data. Part I - A basic framework founded on principal component analysis (PCA), partial least squares (PLS), and statistical molecular design (SMD). Mol Divers 10:169–186. https://doi.org/10.1007/s11030-006-9024-6
35. European Commission JRC-IHCP QSAR Model Reporting Format
36. Pastor M, Quintana J, Sanz F (2018) Development of an infrastructure for the prediction of biological endpoints in industrial environments. Lessons learned at the eTOX project. Front Pharmacol 9:1–8. https://doi.org/10.3389/fphar.2018.01147

Figures

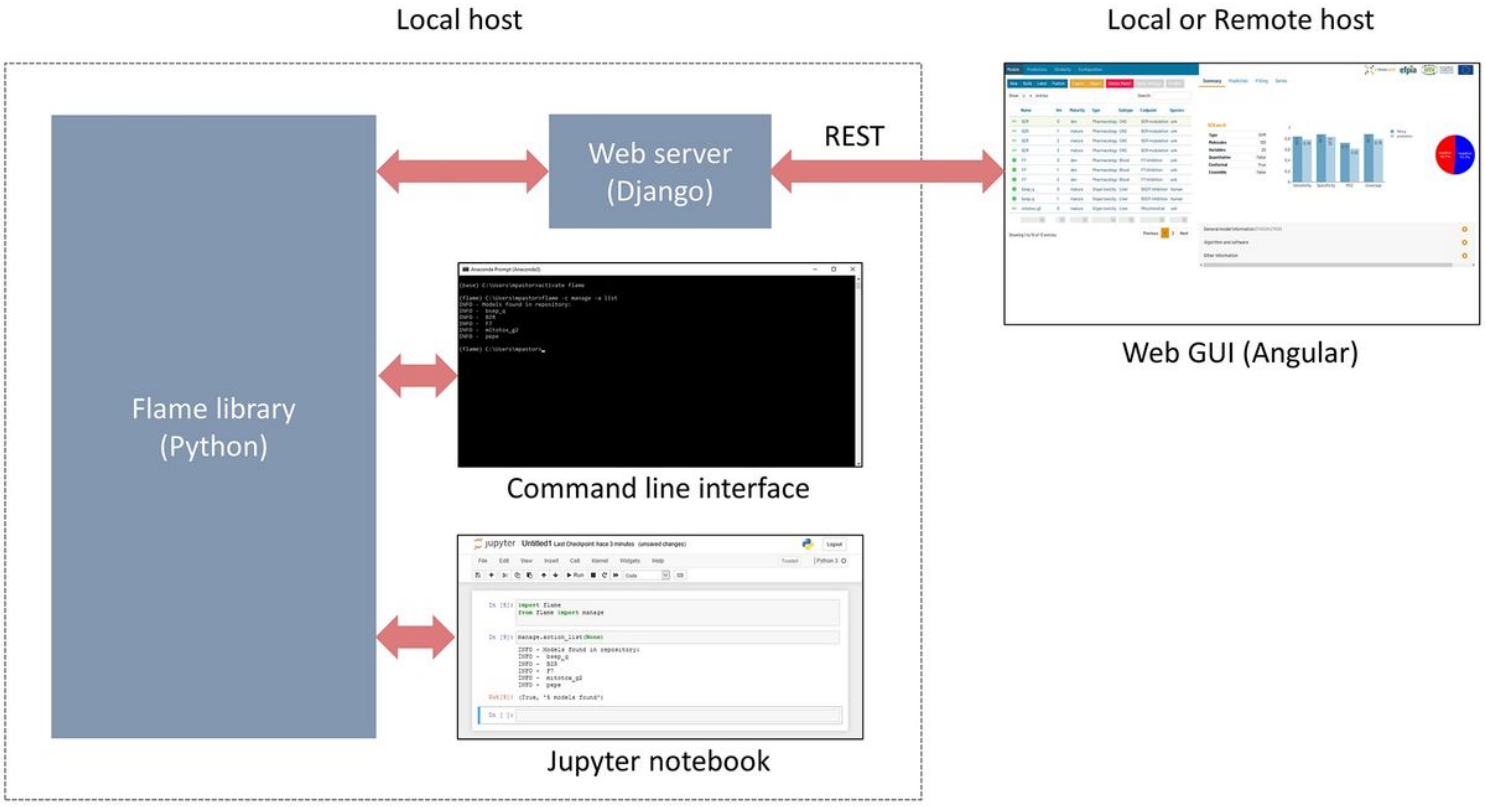


Figure 1

Flame architecture

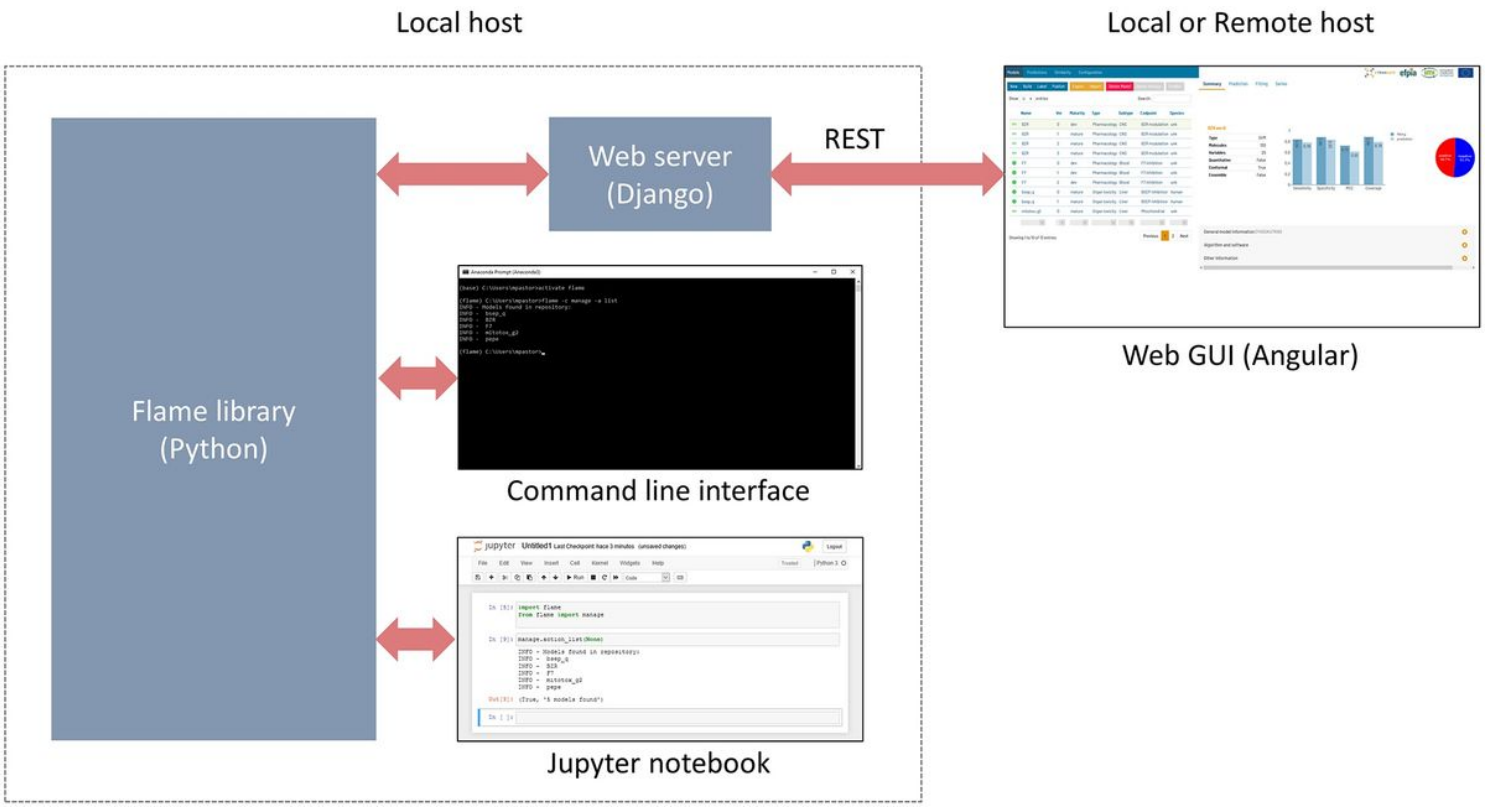


Figure 1

Flame architecture

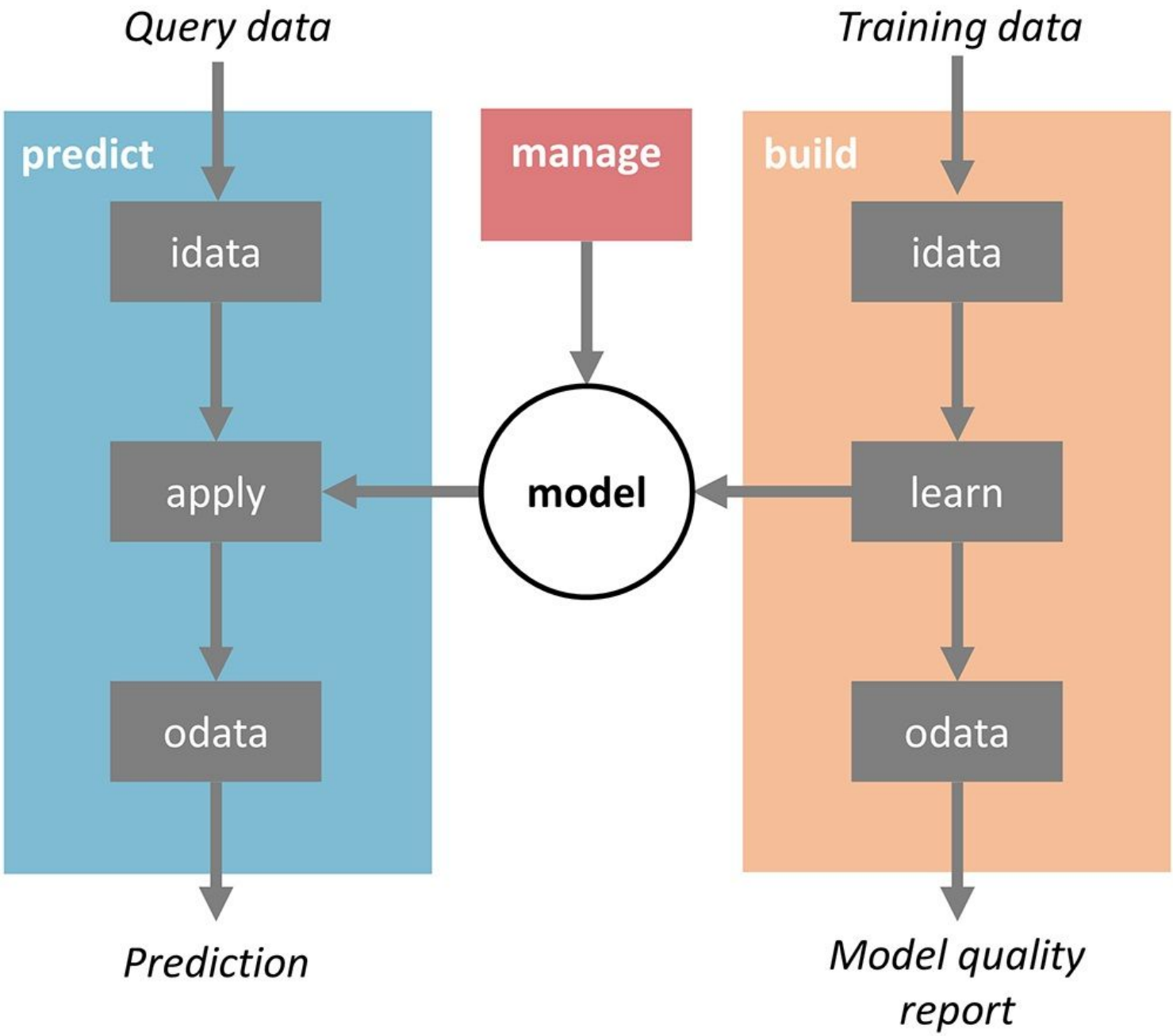


Figure 2

Overview of the main workflows implemented natively in Flame; predict and build. Boxes represents Python classes carrying out specific workflow tasks. As can be seen, some objects (idata, odata) are reused in both workflows, guaranteeing that the same code is used at model building and prediction.

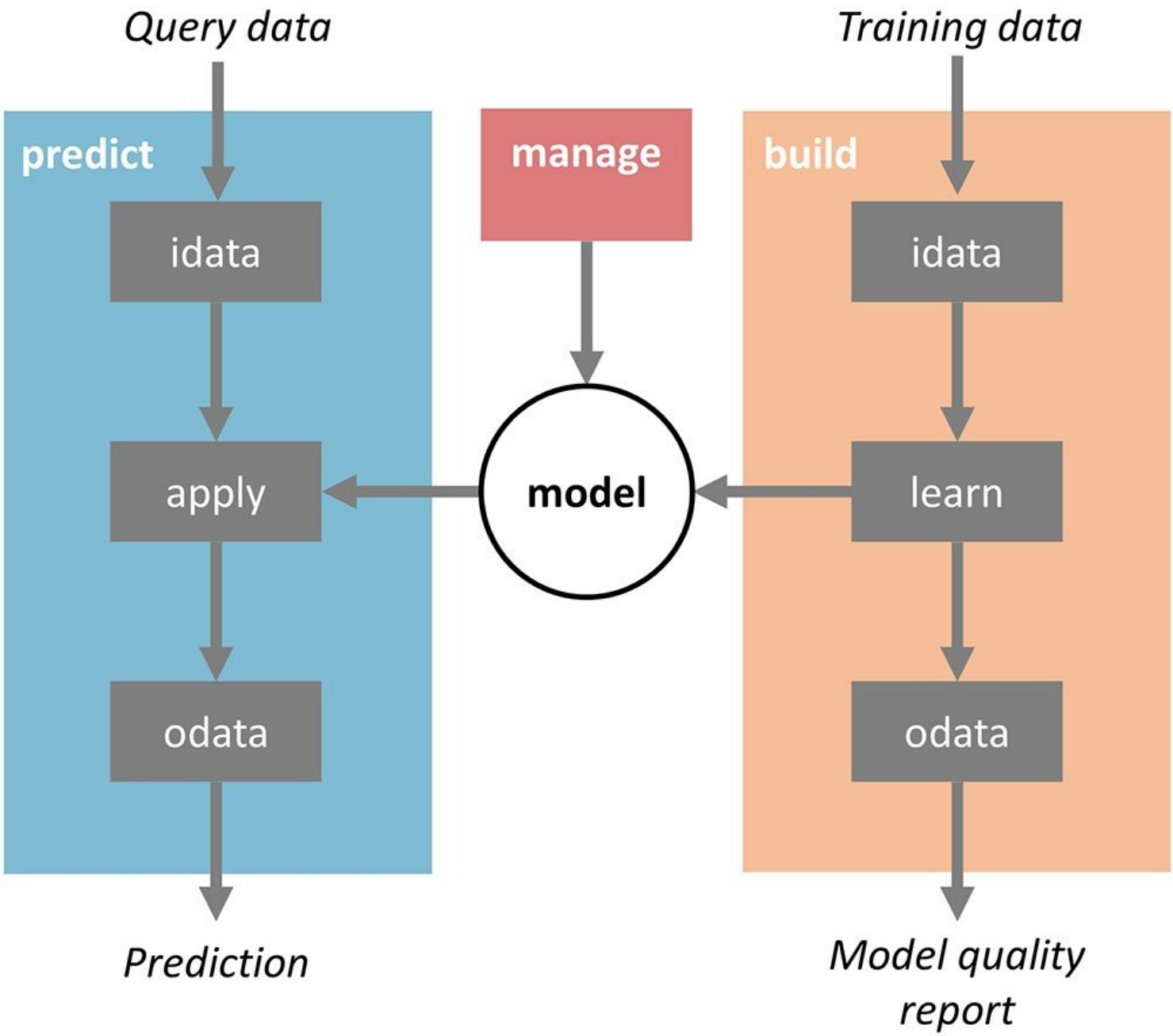


Figure 2

Overview of the main workflows implemented natively in Flame; predict and build. Boxes represents Python classes carrying out specific workflow tasks. As can be seen, some objects (idata, odata) are reused in both workflows, guaranteeing that the same code is used at model building and prediction.

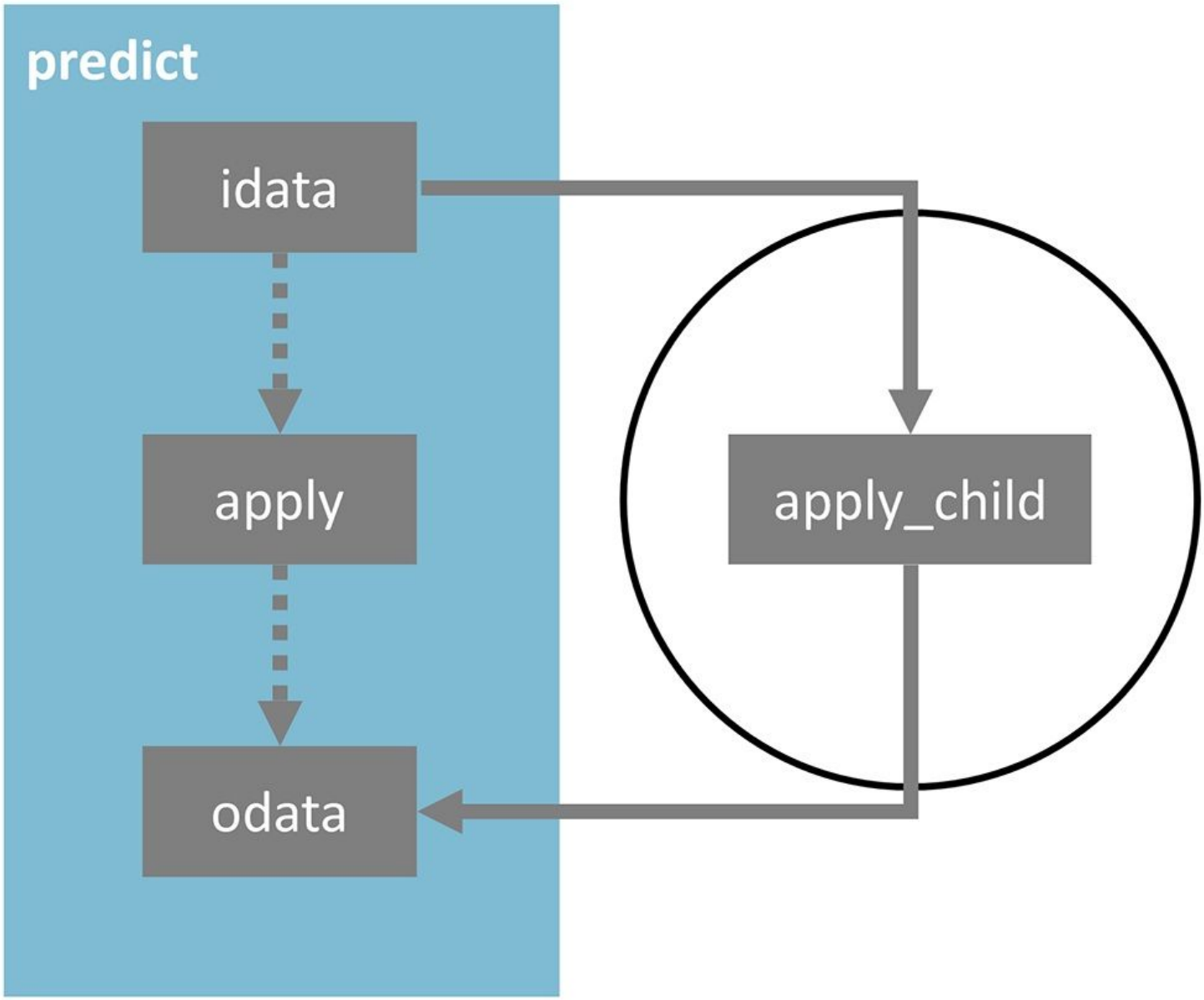


Figure 3

OOP method overriding. Models incorporate children instances of the main low-level classes (see Table 2). By default, the children are empty, and the parent class code is run, but advanced users can edit the code and override the parent class methods to customize the workflow.

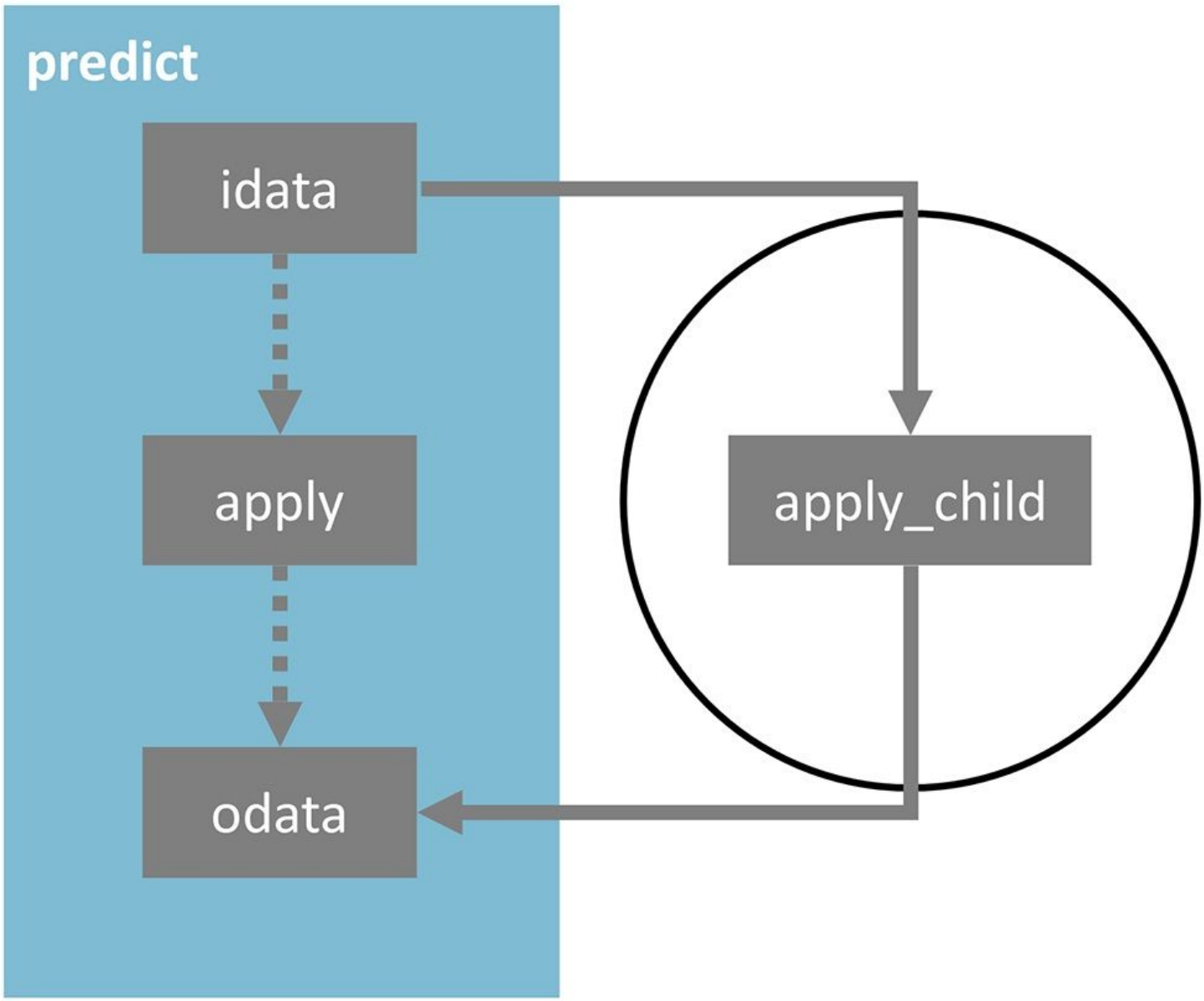


Figure 3

OOP method overriding. Models incorporate children instances of the main low-level classes (see Table 2). By default, the children are empty, and the parent class code is run, but advanced users can edit the code and override the parent class methods to customize the workflow.

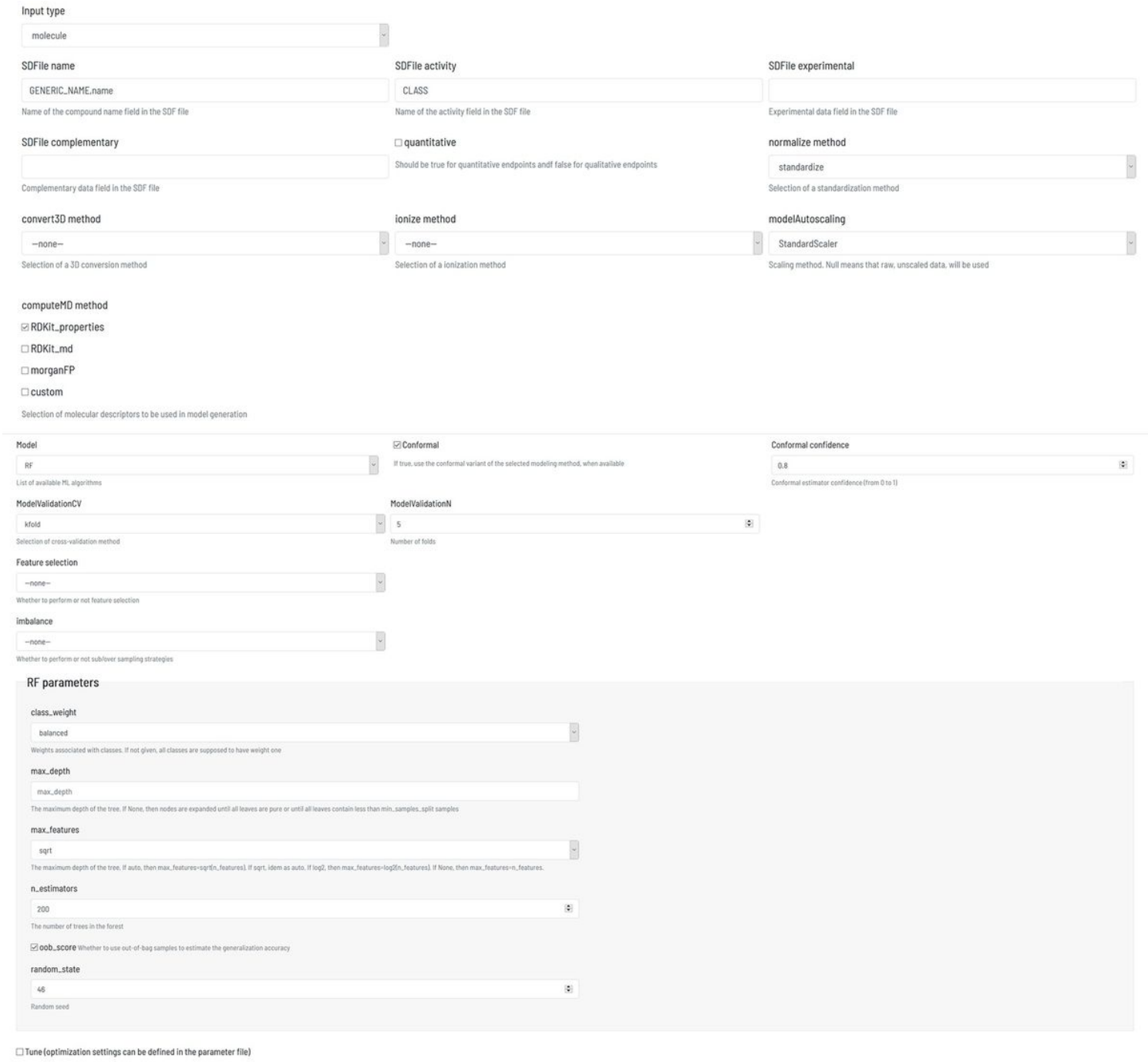


Figure 4

Dialogue used to define the model building workflow methods and parameters (simplified)

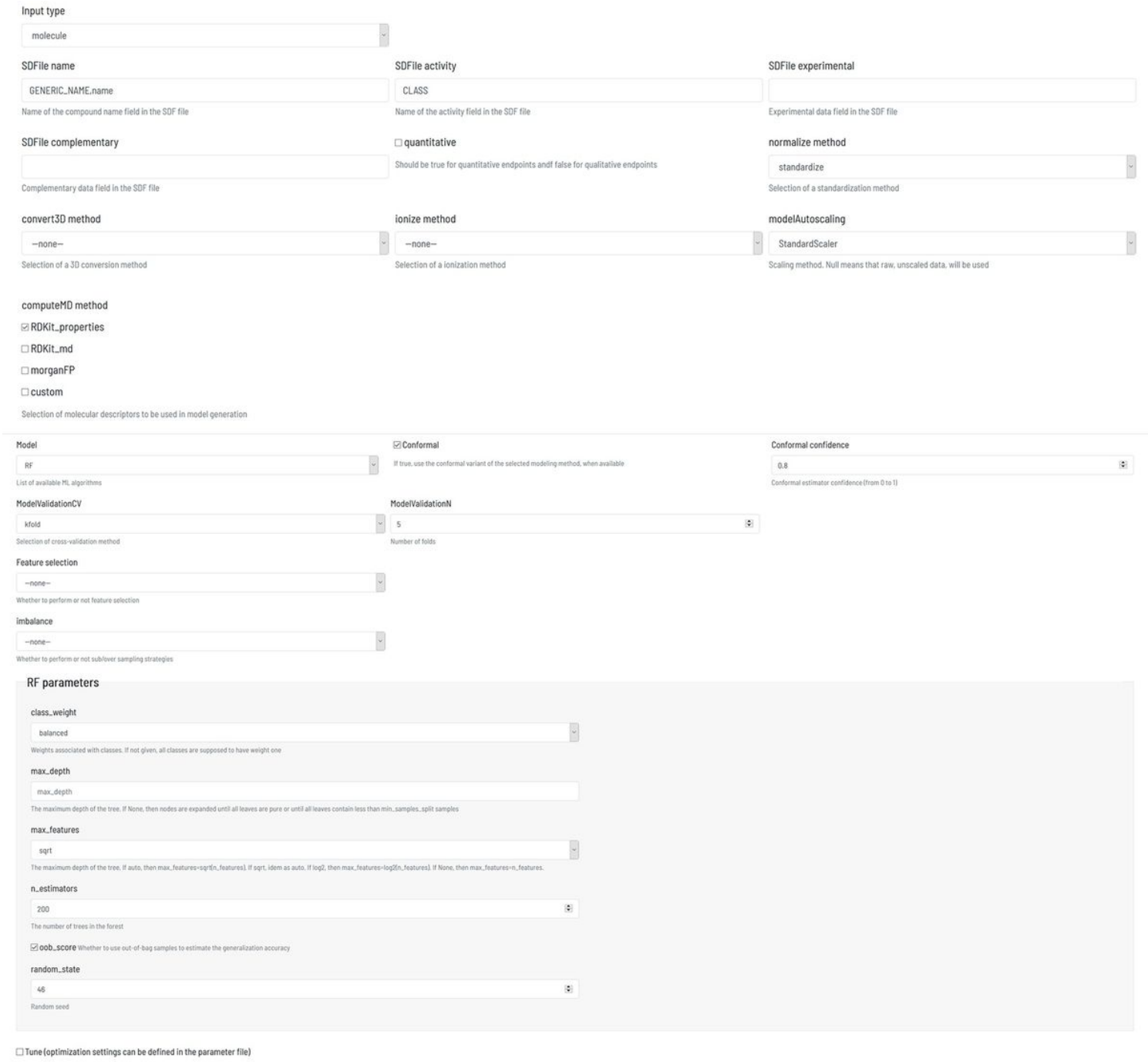


Figure 4

Dialogue used to define the model building workflow methods and parameters (simplified)

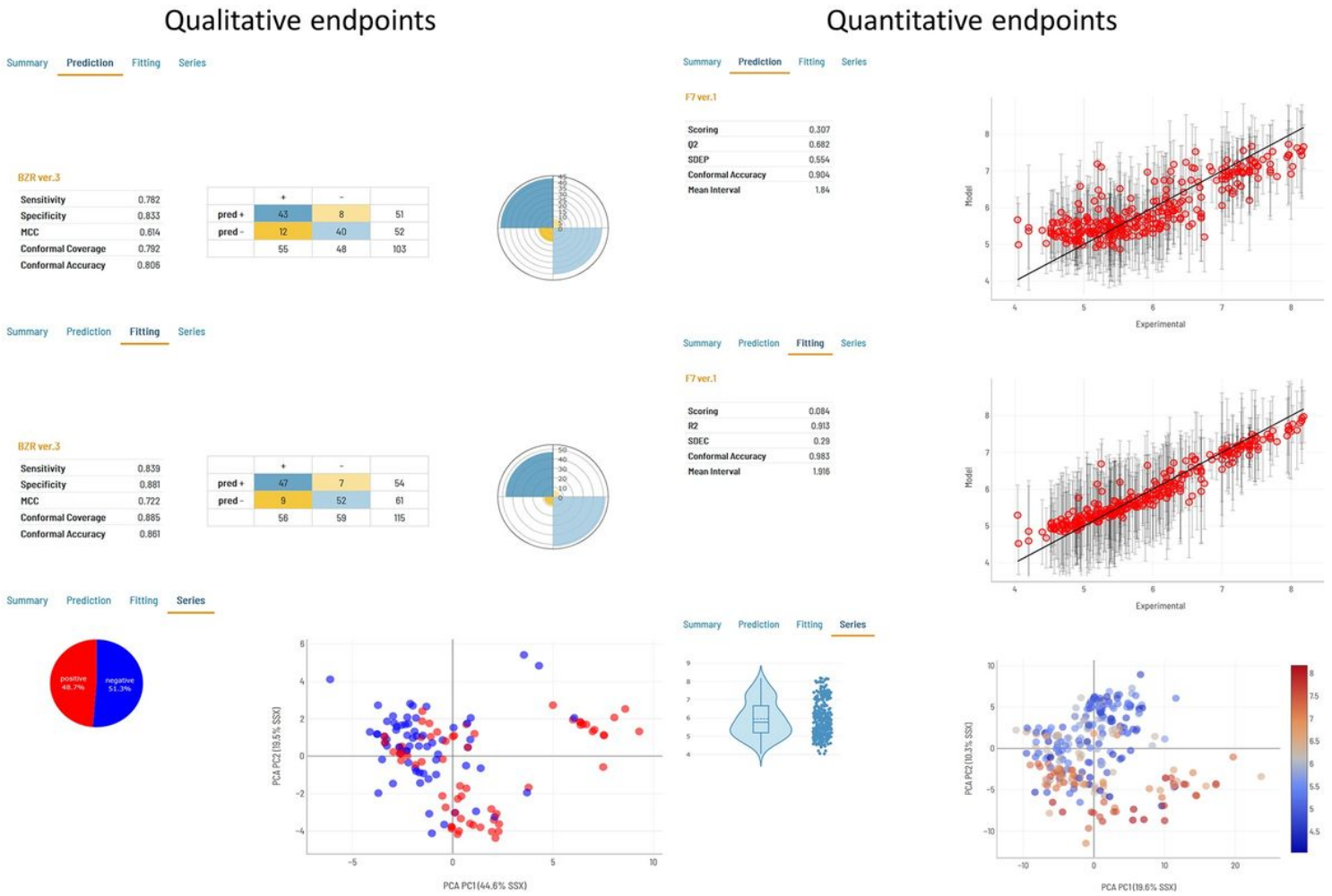


Figure 5

Model output for qualitative (left) and quantitative (right) endpoints

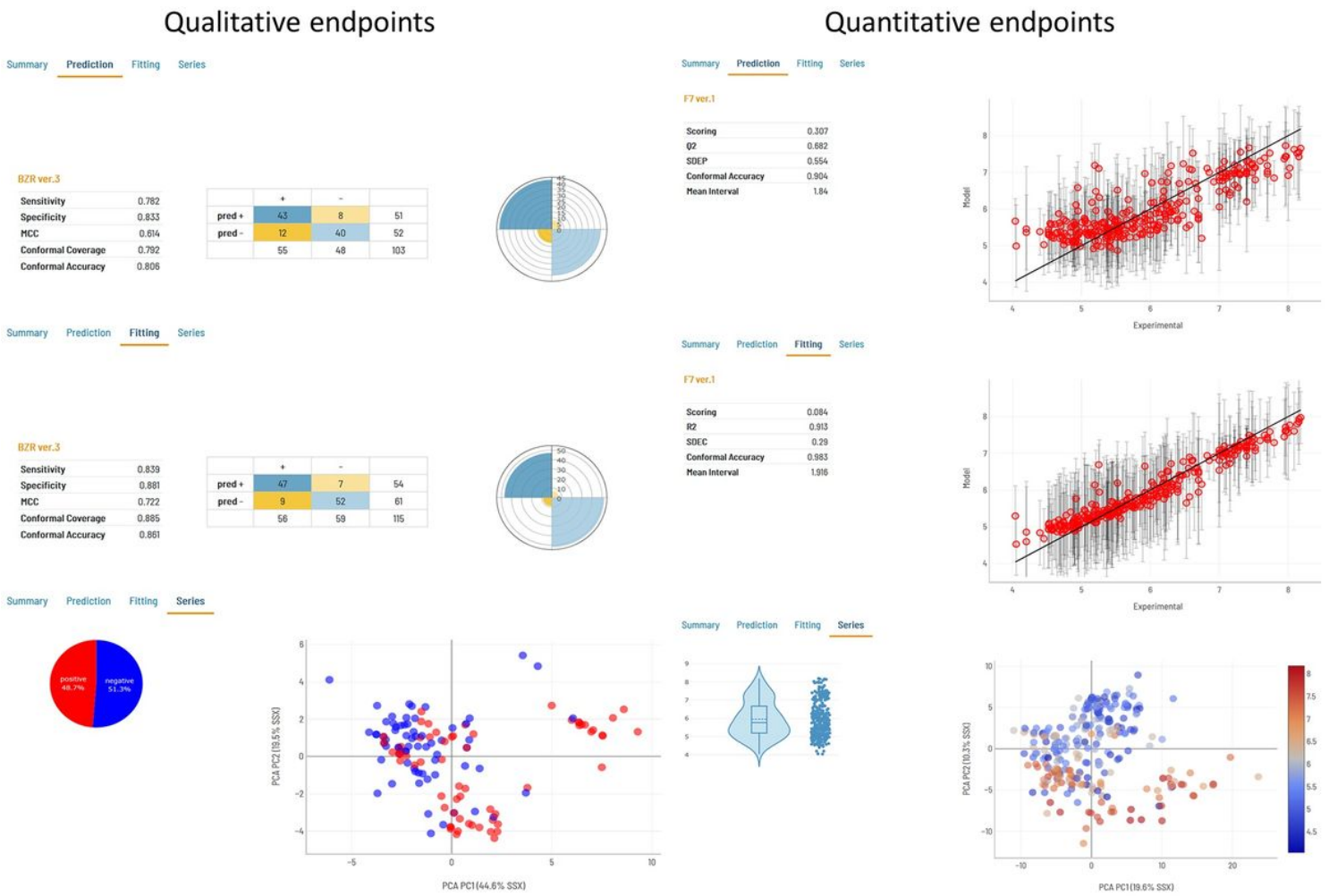


Figure 5

Model output for qualitative (left) and quantitative (right) endpoints

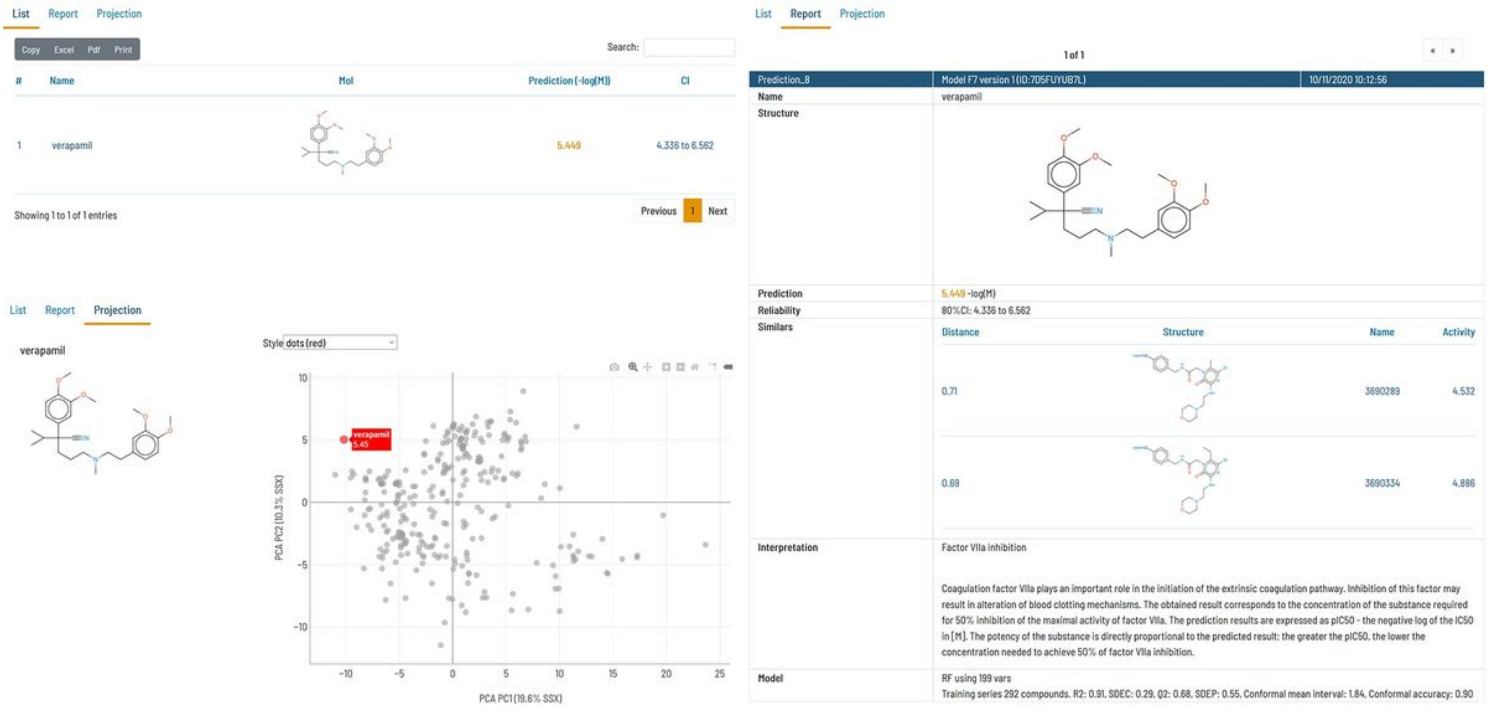


Figure 6

Representation of the model prediction results in the Flame GUI.

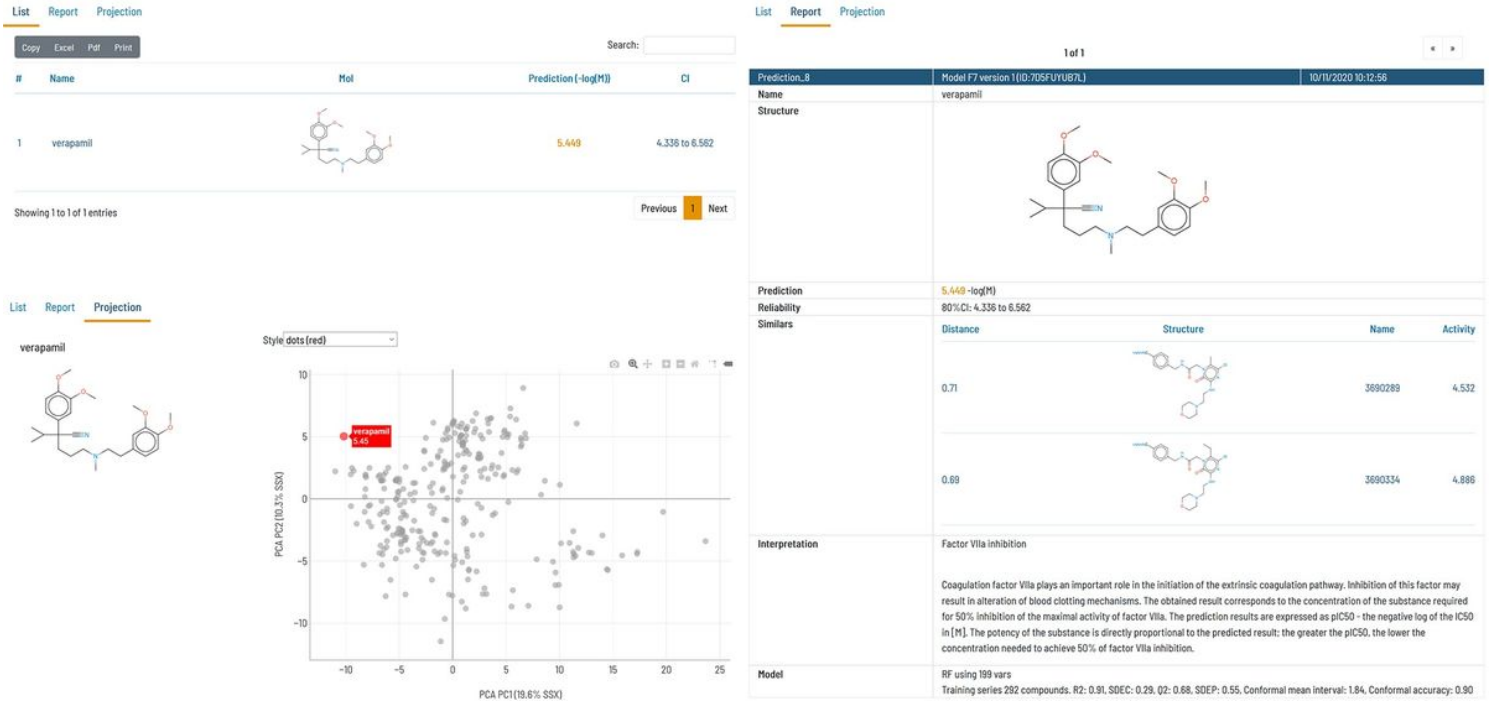


Figure 6

Representation of the model prediction results in the Flame GUI.

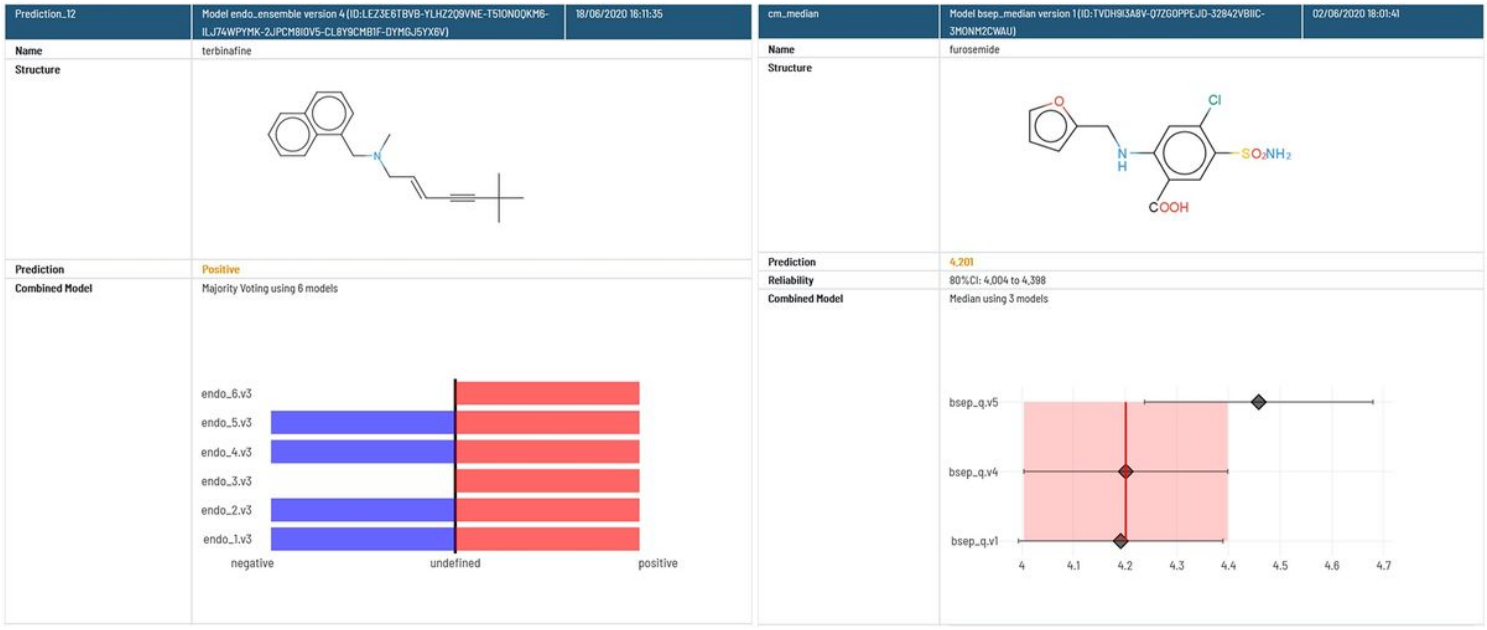


Figure 7

Visualization of prediction results obtained with ensemble models for qualitative (left) and quantitative (right) endpoints.

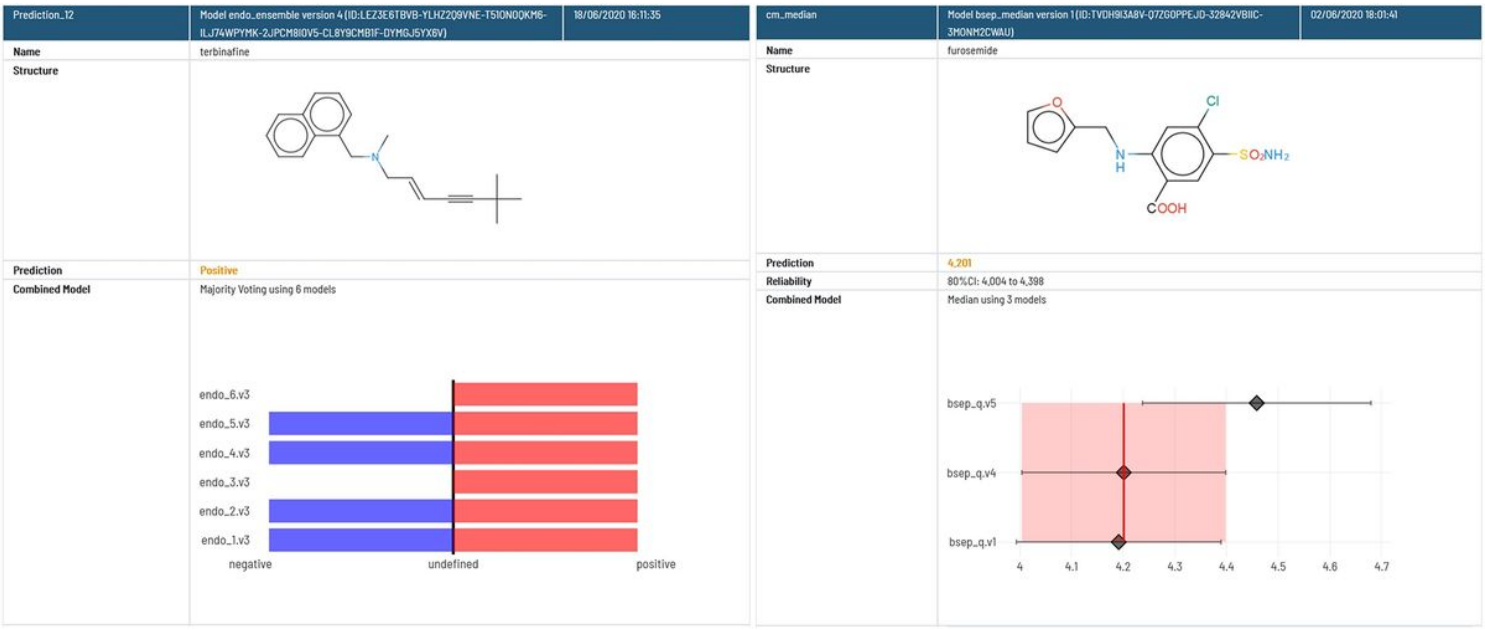


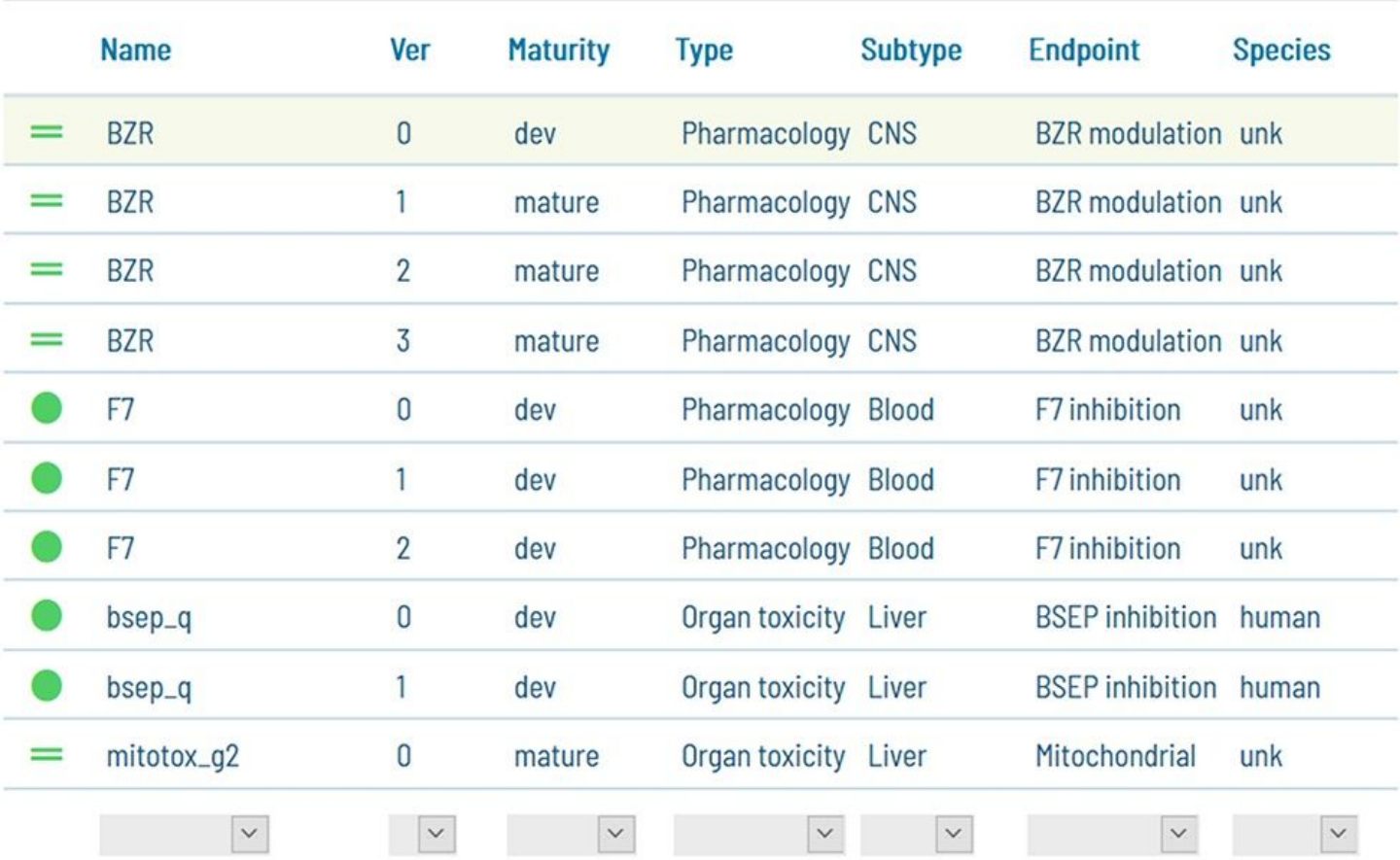
Figure 7

Visualization of prediction results obtained with ensemble models for qualitative (left) and quantitative (right) endpoints.

Name	Ver	Maturity	Type	Subtype	Endpoint	Species
BZR	0	dev	Pharmacology	CNS	BZR modulation	unk
BZR	1	mature	Pharmacology	CNS	BZR modulation	unk
BZR	2	mature	Pharmacology	CNS	BZR modulation	unk
BZR	3	mature	Pharmacology	CNS	BZR modulation	unk
F7	0	dev	Pharmacology	Blood	F7 inhibition	unk
F7	1	dev	Pharmacology	Blood	F7 inhibition	unk
F7	2	dev	Pharmacology	Blood	F7 inhibition	unk
bsep_q	0	dev	Organ toxicity	Liver	BSEP inhibition	human
bsep_q	1	dev	Organ toxicity	Liver	BSEP inhibition	human
mitotox_g2	0	mature	Organ toxicity	Liver	Mitochondrial	unk

Figure 8

The models present in the model repository are shown in the GUI as a model list. Items can be sorted, browsed, and searched by text terms. Models include user-defined labels which can be used to filter the list content



Name	Ver	Maturity	Type	Subtype	Endpoint	Species
BZR	0	dev	Pharmacology	CNS	BZR modulation	unk
BZR	1	mature	Pharmacology	CNS	BZR modulation	unk
BZR	2	mature	Pharmacology	CNS	BZR modulation	unk
BZR	3	mature	Pharmacology	CNS	BZR modulation	unk
F7	0	dev	Pharmacology	Blood	F7 inhibition	unk
F7	1	dev	Pharmacology	Blood	F7 inhibition	unk
F7	2	dev	Pharmacology	Blood	F7 inhibition	unk
bsep_q	0	dev	Organ toxicity	Liver	BSEP inhibition	human
bsep_q	1	dev	Organ toxicity	Liver	BSEP inhibition	human
mitotox_g2	0	mature	Organ toxicity	Liver	Mitochondrial	unk

Figure 8

The models present in the model repository are shown in the GUI as a model list. Items can be sorted, browsed, and searched by text terms. Models include user-defined labels which can be used to filter the list content

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- BZR.yaml
- BZR.yaml
- modeldocumentationGUI.pdf
- modeldocumentationGUI.pdf