AUTOML - AUTO-PYTORCH

Marcel Witas
Warsaw University of Technology
Faculty of Mathematics and Information Science

Adam Frej
Warsaw University of Technology
Faculty of Mathematics and Information Science

Łukasz Tomaszewski
Warsaw University of Technology
Faculty of Mathematics and Information Science

June 27, 2022

ABSTRACT

Machine Learning becomes more popular and so does automated machine learning. In this paper, we describe Auto-PyTorch, an Auto Deep Learning framework, which offers classification and regression on tabular data and image classification, and also our own implementation of an auto machine learning pipeline, which was made for binary classification tasks. If it comes to Auto-PyTorch, we wanted to find out, what kind of preprocessing is necessary for the framework to work and what results can we get. We tested the framework on binary classification tasks from OpenML. Even though we had limited time and compute resources, we managed to obtain results, which were comparable with results of other state-of-the-art frameworks. As the next step, we build our own auto machine learning pipeline. We compared our pipeline with Auto-Pytorch on a binary classification task. The results of both frameworks were similar, but the main difference was the evaluation time. This test showed us, that obtaining a good result is not difficult, but achieving good anytime performance is a challenging task.

Keywords Machine Learning · Deep Learning · Automated Machine Learning · Auto-Pytorch · OpenML Benchmark

1 Introduction

Recently machine learning technology (ML) has gained in popularity and so have done automated machine learning (AutoML) [1]. The goal of AutoML is to provide ML algorithms in easy to use way with minimal lines of code required. There are many AutoML frameworks available, e.g. *AutoGluon* [2], *AutoKeras* [3], *Auto-sklearn* [4]. Achieving good anytime performance is a challenging task, so different frameworks use different approaches to this problem. While most frameworks focus on traditional ML pipelines, some of them try to include Deep Learning (DL). One of these is *Auto-Pytorch* [5] which tries to combine ML algorithms with DL architecture. We try to review the framework and recreate the results of AutoML benchmark [6] that authors conducted. Using official documentation and repository with source code we executed the framework and obtained the results. We have also created our own AutoML pipeline which in much simpler form also combines DL with traditional ML. We want to compare it to Auto-Pytorch. The aim of this paper is to introduce AutoML and Auto-Pytorch. We wanted to reproduce Auto-PyTorch article [5] and evaluate the framework on tabular data.

2 Auto-PyTorch

2.1 Framework description

Auto-PyTorch [5] is an Auto Deep Learning framework, which offers classification and regression on tabular data and image classification. It optimizes Neural Network architectural parameters and training hyperparameters, by using

multi-fidelity optimization. It relies on *PyTorch* [7] as the DL framework. Auto-PyTorch implements and optimizes DL pipeline, which includes data preprocessing, neural architecture, network training techniques and regularization techniques. This framework offers warmstarting by sampling configurations from portfolio. It also ensures an automated ensemble selection.

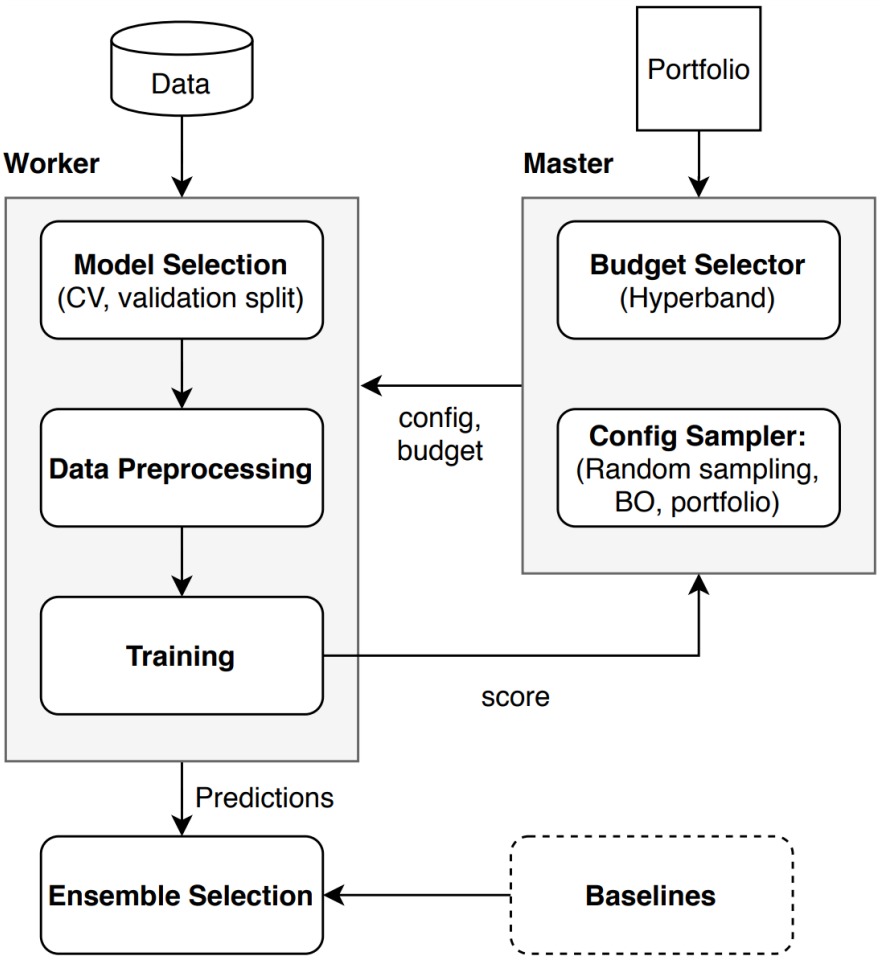

Figure 1: Auto-PyTorch workflow (figure taken from article [5])

Auto-PyTorch contains many hyperparameters. User can change preprocessing options (e.g. encoding, imputation), architectural hyperparameters (e.g. network type, number of layers) and training hyperparameters (e.g. learning rate, weight decay). Hyperparameters have a hierarchy, top-level hyperparameters can change sub-level hyperparameters.

To achieve good anytime performance Auto-PyTorch uses *BOHB* [8], which is a combination of *Bayesian Optimalization* [9] and *Hyperband* [10]. It has been shown, that BOHB outperforms BO and HB on many tasks and it is also up to 55 times faster than *Random Search* [11]. BOHB uses an outer and an inner loop like HB. In the inner loop configurations are evaluated on lower budget. Then well-performing configurations are being promoted to higher budgets via *SuccessiveHalving* [12]. Budgets are being calculated in the outer loop. BOHB fits a kernel density estimator (KDE) as a probabilistic model to the observed performance data. This allows it to find promising areas in the configuration space. Since the configurations on smaller budgets are evaluated first, the KDE's on smaller budgets are available earlier and can guide search until better models on higher budgets are available.

Auto-PyTorch uses a master-worker architecture. We can see the scheme on Figure 1. Thanks to this, it can use additional compute resources.Auto-PyTorch uses user-defined splits or automated splitting or cross-validation with any iterator from *scikit-learn* [13]. The score of a model is evaluated via a predefined or user specified metric.Auto-PyTorch trains many models and then ensembles them. Ensembling is inspired by *auto-sklearn* [4]. It starts on an empty set and adds to it, the model, which gives the biggest performance improvement. This process lasts until the set reaches a predefined size. Because one model can be added to the set multiple times, every model gets a weight. BOHB start

from zero for each new task. To improve performance, Auto-PyTorch uses a *warmstart* [14]. BOHB first iteration is started with a set of configurations, which cover a set of meta-training datasets.

2.2 Our contribution to preprocess data

Since Auto-PyTorch implements and automatically tunes the full Deep Learning pipeline, including data preprocessing, there is no need to do a lot of preprocessing. However, framework accepts only numerical and categorical types of data. Therefore, we changed columns with the object type to categorical. Such preprocessing turned out to be enough for Auto-PyTorch to work.

3 Evaluation results

To evaluate performance of Auto-PyTorch, we implemented function that prepares any tabular dataset describing a prediction problem to use Auto-PyTorch. Subsequently, we selected 22 binary classification datasets from An Open Source AutoML Benchmark [6]. The datasets vary in the number of samples and features by orders of magnitude, and vary in the occurrence of numeric features, categorical features and missing values. We used the same metric like in benchmark, area under the receiver operator curve (AUROC). We also used 10-fold cross-validation with identical folds.

As we had limited time, we had to evaluate tasks on lower budget, than they did in the benchmark [6]. It was difficult to choose optimal time limits, because Auto-PyTorch require not only total time limit, but also maximum time for one model (which is by default two times smaller than total time, however it sometimes allows too little estimators to train). Eventually we used different times for different datasets. They are different to those used in benchmark [6], where 1 and 4 hours per fold were applied. We trained 7 datasets using 6 minutes per fold, 40 seconds per model, 12 datasets using 30 minutes per fold, 900 seconds per model, and 3 datasets using 1 hour per fold, 450 seconds per model.

The results of Auto-PyTorch on every fold can be seen on Figure 2. The variance of the per-fold scores can be quite large. To compare it with frameworks tested in benchmark [6] (1 hour per fold), we have to look at Figure 3. Despite lower time limits, Auto-PyTorch perform worse than all other frameworks only on one dataset (*christine*). On most datasets Auto-PyTorch performed better than some frameworks, but was never the best. Note that Auto-PyTorch uses Deep Learning, which might be advantage comparing to other frameworks.

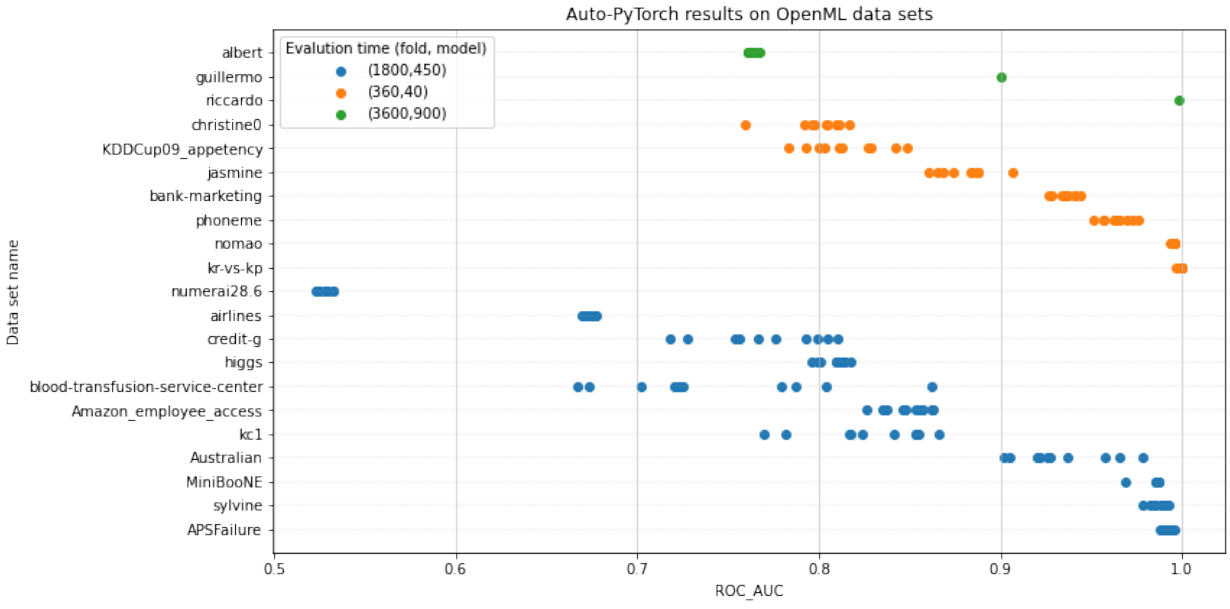

Figure 2: Auto-PyTorch results on binary classification tasks from OpenML benchmark [6]. Due to technical reasons guillermo and riccardo were trained on one fold.

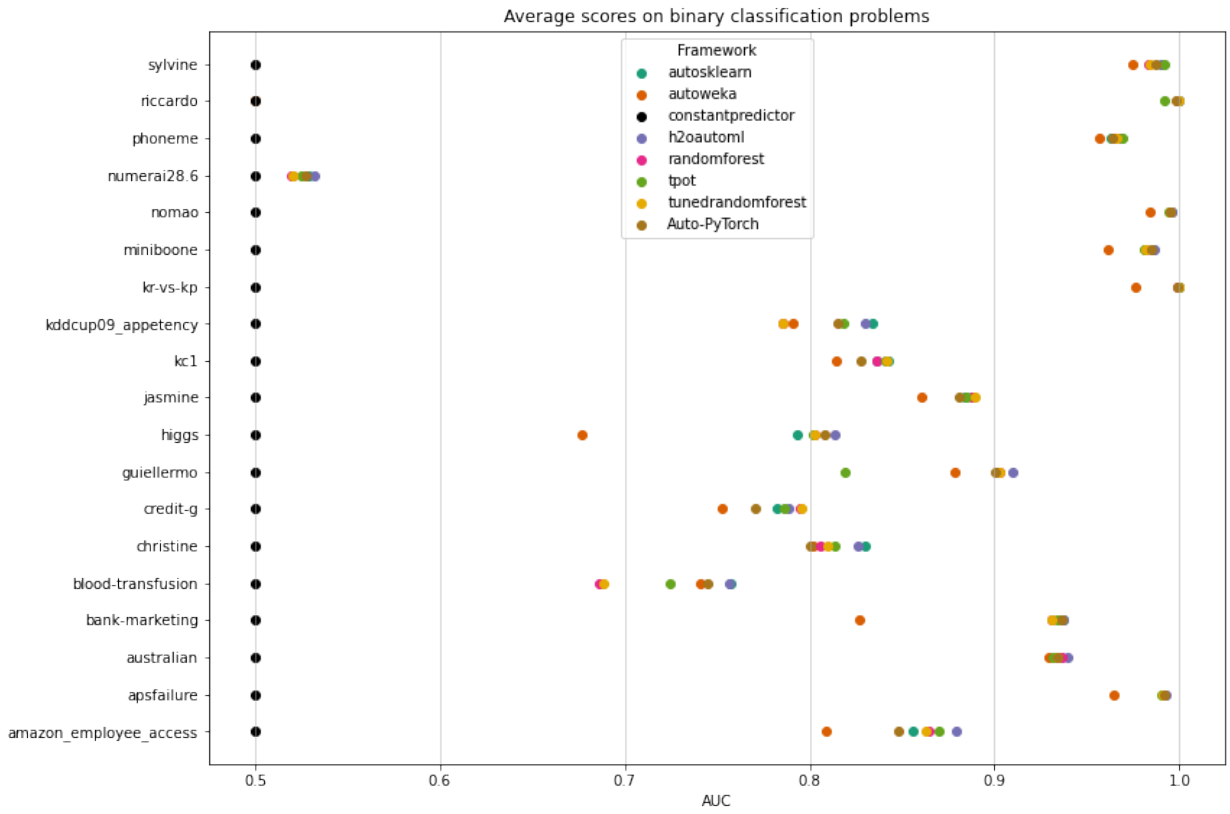

Figure 3: Average AUROC results of frameworks tested in benchmark [6] (1h per fold) compared to AutoPyTorch

We explored, which models were involved in the ensembling most frequently. There were huge differences between folds. While in folds 2-10 the most common models where traditional models (e.g. *CBLearner*, *ETLearner*), the most common models in first folds were based on Deep Learning. Even though AUC score were similar. We can't draw clear conclusions about the reasons, but it might be connected with some memory problems. However, from traditional models, CatBoost, ExtraTrees and Random Forest were most often used. Models based on deep neural networks were used only in first folds. The estimators with number of folds in which they were used are shown on Figures 4 and 5.

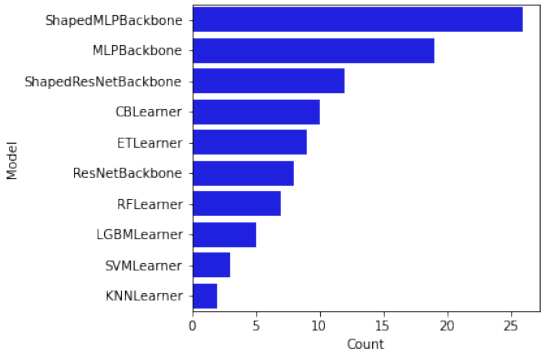

Figure 4: Most common models in fold 1.

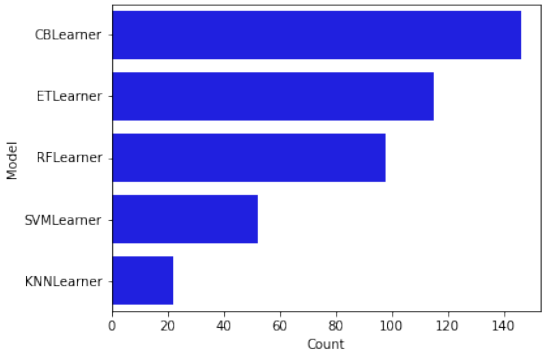

Figure 5: Most common models in folds 2-10.

4 Own implementation

4.1 Description

Our pipeline is made for binary classification tasks. As parameters, it takes a training set, a variable to predict and a set on which we want to make predictions. It performs a simple preprocessing, which starts with removal of duplicates. Next any missing values are being imputed. To do this, it uses *KNNImputer* [15] with number of neighbours set to 1. Thanks to this, our pipeline will fill the gaps with values, that are already present in the set. Categorical variables are encoded using *OneHotEncoder* [15]. The pipeline also drops columns with variance under 0.2. At the end of preprocessing it scales variables to be in range [0,1]. After preprocessing, our pipeline trains three classic ML models: *SVC* [15], *XGBoost* [16] and *RandomForestClassifier* [15]. To find the best hyperparameters, it launches *Bayesian optimization* [9] on predefined parameters space. Then it trains two models, which use Deep Learning. Those models were taken from *Keras* package [17]. They are two neural networks, one with sigmoid activation function and the other with ReLU. Next, it takes all of the five models and ensembles them using *Soft Voting* [15] to make the final model. At the end, the pipeline fits the final model on the training set and the predicted variable, and counts the probabilities for the test set. The pipeline returns the final model and probabilities.

4.2 Results

To find out, how good our pipeline is, we run it on a set [1], which was given to us on lecture and compared the results with Auto-PyTorch results.

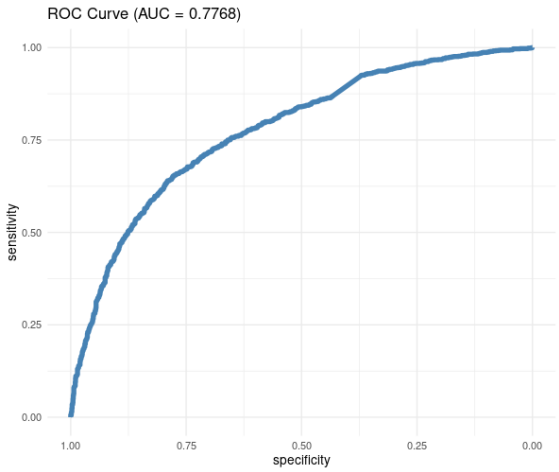

Figure 6: ROC curve - our pipeline

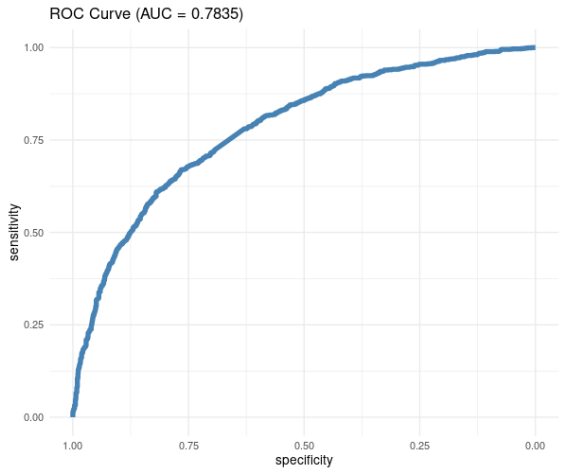

Figure 7: ROC curve - Auto-PyTorch

As we can see on Figure 6 and Figure 7, AUC [18] of Auto-PyTorch prediction was higher than AUC of our pipeline prediction, but the difference was not big. Nevertheless, our pipeline needed one hour to evaluate, when Auto-PyTorch needed only ten minutes. This test showed us, that obtaining a good result is not difficult, but achieving good anytime performance is a challenging task.

5 Discussion & Conclusion

This paper reviewed and evaluated Auto-Pytorch. It described framework architecture and explained how particular components contribute to overall performance. We checked the theory by conducting binary OpenML benchmark using AUROC metric. In general the results met the expectations. Despite having less computing power and giving less training time that original authors we outperformed other frameworks. It proves that BOHB compared with Deep Learning is a very efficient architecture.

[1] `https://community.fico.com/s/explainable-machine-learning-challenge?tabset-158d9=2`

We also introduced our own AutoML framework containing preprocessing, Bayesian Optimization performed on three classic ML models and two DL Neural Networks. We evaluated it on a single dataset and compared the result with Auto-Pytorch. Obtained scores were very similar but our pipeline needed much more time to get the prediction.

In conclusion we hope that AutoML technology will continue its development as it clearly has a big potential. There is still room for more improvements as new frameworks and articles are still published.

References

1. F. Hutter, L. Kotthoff, and J. Vanschoren. Automatic Machine Learning: Methods, Systems, Challenges. Challenges in Machine Learning. *Springer*, 2019.

2. Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

3. Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019.

4. Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine Learning. *Advances in Neural Information Processing Systems*, 28, 2015.

5. Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL. 2020. doi:10.48550/ARXIV.2006.13799.

6. Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An Open Source AutoML Benchmark. 2019. URL `http://arxiv.org/abs/1907.00909`.

7. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An Imperative Style, High-performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 2019.

8. Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

9. Jonas Mockus. Application of Bayesian Approach to Numerical Methods of Global and Stochastic Optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.

10. Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

11. James Bergstra and Yoshua Bengio. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research*, 13(2), 2012.

12. Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Artificial Intelligence and Statistics*, pages 240–248. PMLR, 2016.

13. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

14. Matthias Feurer, Jost Springenberg, and Frank Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

15. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

16. Tianqi Chen and Carlos Guestrin. Xgboost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

17. François Chollet et al. Keras. `https://keras.io`, 2015.

18. Jorge M Lobo, Alberto Jiménez-Valverde, and Raimundo Real. AUC: A Misleading Measure of the Performance of Predictive Distribution Models. *Global Ecology and Biogeography*, 17(2):145–151, 2008.