# Supplementary Materials for Niseko: a Large-Scale Meta-Learning Dataset

**Zeyuan Shang [1], Emanuel Zgraggen [1], Philipp Eichmann [2], Tim Kraska [1]**
Massachusetts Institute of Technology[1], Brown University[2]
`{zeyuans, emzg, kraska}@mit.edu, peichman@cs.brown.edu`

## A Related Work

There have been several studies for an overview of meta-learning techniques [28, 30]. With the prior tasks and the evaluations (e.g., accuracy or time) of some learning algorithms, there have been several studies aiming to find some promising configurations of learning algorithms given a new task. Through *surrogate models* which are built from previous tasks, we can measure task similarity and thus apply Bayesian Optimization [4] to find the next promising model for the new dataset. Wistuba et al. [32] train Gaussian Processes as surrogate models for prior tasks and the new task, and measure their similarity based on the means. Feurer et al. [11] combine the predictive distributions of Gaussian processes into a Gaussian process as the surrogate model.

Another way for connecting prior tasks with the current new task is to jointly learn a task description, e.g., Swersky et al. [25] propose multi-task learning given a set of pre-defined similar tasks, which transfers knowledge between each pair of task by learning a joint model. Leite et al. [16] predict the shapes of learning curves by adapting the nearest complete curves to the partial curve.

Besides using previous evaluations of learning algorithms, some studies focus on the properties of the task itself, e.g., number of instances, class entropy [28]. Feurer et al. [10] propose a set of meta-features to quantify similarity between datasets and thus transfer Bayesian Optimization priors between them. Some studies introduce collaborative filtering techniques along with meta-features to recommend promising models [22, 33]. Other studies learn a meta-model based on meta-features of tasks to predict the ranking of learning algorithms [3, 23].

Other than the evaluations and tasks, we are also able to learn from prior learning algorithms, e.g., the structure of pipelines and their parameters, in other words, we learn how to train a new model for a new task from previous models. One of the most widely-used technique is *transfer learning* [26], and it achieves notable progress on image classification tasks [7, 20]. Andrychowicz et al. [1] propose a learned optimizer for neural networks by training an LSTM on some prior tasks to learn how to update weights. Another interesting topic is *few-shot learning*, which trains a deep neural network given a few training examples and some previous evaluations of similar tasks with abundant training data [14, 21, 18].

Meta-learning has also been an important building block for an AutoML system to warm start the search of pipelines, e.g., Auto-sklearn uses meta-learning to find some good initial models [9].

However, there have been few public datasets to share the meta-data and also experiment with the new meta-learning techniques for meta-learning researchers. *META-DATASET* [27] is a benchmark for training and evaluating only few-shot classifiers. Data science websites like *Kaggle* have lots of public datasets, however, the solutions uploaded by people are either text description of their methods or IPython notebook, which are hard to use in large-scale. *OpenML* [29] is a place for machine learning researchers to share and organize data, however, it only exposes predictive models (thus pre-processing operations are ignored), and users are not able to run the models locally. Niseko addresses the issues with those datasets by exposing the structure of pipelines, their running traces,

3rd Workshop on Meta-Learning at NeurIPS 2019 NeurIPS 2019, Vancouver, Canada.

datasets/tasks with easy-to-use APIs. Users are also able to run these pipelines as they can be converted into an executable Python script to reproduce results.

## B Search Space

| name | source | #$\lambda$ |
|---|---|---|
| SVC | sklearn | 7 |
| LinearSVC | sklearn | 5 |
| LogisticRegression | sklearn | 4 |
| SGDClassifier | sklearn | 6 |
| RandomForestClassifier | sklearn | 6 |
| GaussianNB | sklearn | - |
| KNeighborsClassifier | sklearn | 2 |
| BaggingClassifier | sklearn | 2 |
| ExtraTreesClassifier | sklearn | 6 |
| GradientBoostingClassifier | sklearn | 9 |
| XGBClassifier | xgboost | 5 |
| LinearDiscriminantAnalysis | sklearn | 1 |
| QuadraticDiscriminantAnalysis | sklearn | 2 |
| DecisionTreeClassifier | sklearn | 4 |
| LGBMClassifier | lightgbm | 4 |

Table 1: classification algorithms: name, source and number of hyper-parameters

| name | source | #$\lambda$ |
|---|---|---|
| SVR | sklearn | 8 |
| LinearSVR | sklearn | 5 |
| Ridge | sklearn | 1 |
| SGDRegressor | sklearn | 7 |
| RandomForestRegressor | sklearn | 6 |
| GaussianProcessRegressor | sklearn | 1 |
| KNeighborsRegressor | sklearn | 2 |
| ExtraTreesRegressor | sklearn | 6 |
| GradientBoostingRegressor | sklearn | 9 |
| XGBRegressor | xgboost | 5 |
| ARDRegression | sklearn | 7 |
| DecisionTreeRegressor | sklearn | 4 |
| LGBMRegressor | lightgbm | 4 |
| RuleFit | rulefit | 2 |

Table 2: regression algorithms: name, source and number of hyper-parameters

Table 1, 2, 3 shows the classification, regression and feature preprocessing algorithms used in Niseko. We also released the hyper-parameter distributions at `https://github.com/niseko-submission/niseko_submission/search_space`.

## C Code, Datasets and Tasks

We have released our code and data at `https://github.com/niseko-submission/niseko_submission`.

## D API

We provide a light-weight Python library to interact, search and explore Niseko's raw meta-learning data. In this API, the context object is the main bridge between the user and the raw data. From it, users can obtain statistical information either for all dataset or for individual ones.

| name | source | #$\lambda$ |
|---|---|---|
| Imputer | sklearn | 1 |
| MinMaxScaler | sklearn | - |
| StandardScaler | sklearn | - |
| RobustScaler | sklearn | - |
| LabelEncoder | sklearn | - |
| OneHotEncoder | sklearn | 1 |
| PCA | sklearn | 2 |
| KernelPCA | sklearn | 5 |
| TruncatedSVD | sklearn | 1 |
| FastICA | sklearn | 3 |
| PolynomialFeatures | sklearn | 3 |
| SelectPercentile | sklearn | 1 |
| GenericUnivariateSelect | sklearn | 2 |
| VarianceThreshold | sklearn | 1 |
| FeatureAgglomeration | sklearn | 3 |
| RBFSampler | sklearn | 2 |
| Normalizer | sklearn | - |

Table 3: preprocessing methods: name, source and number of hyper-parameters

```
import niseko

# initialize the context object
context = niseko.get_context('/niseko_data')
# query all datasets
for ds in context.list_datasets():
    print(ds.dataset_id)
# query a dataset
ds = context.get_dataset_by_id('185_baseball')
# show statistics of this dataset
ds.show_stats()
# output:
# Task Type: CLASSIFICATION
# NumberOfClasses: 3
# NumberOfFeatures: 17
# NumberOfInstances: 1073
# ...
```

Given a dataset, a user can write queries to retrieve associated pipelines and their internals. For example, a user might want to see the models of the top three highest performing pipelines of a particular dataset. Each pipeline can be exported as an executable Python script for further investigation and evaluation.

```
# get pipelines
for pipeline in ds.get_pipelines(order_by='performance', num=3):
    print(pipeline.model)
# output:
# GaussianNB
# export
pipeline.to_script('pipeline.py')
```

There is a detailed documentation of API at `https://github.com/niseko-submission/niseko_submission/blob/master/api/README.md`.

## E Use Case: Model Performance Prediction

Meta-features are used extensively as a similarity metric between datasets in meta-learning techniques to transfer learning algorithms across tasks. Niseko automatically exposes the meta-features used in Auto-sklearn [9] for all datasets. These meta-features can help to quantify the similarity of datasets.

We can use this to, for example, check if a particular model (Xgboost classifier) performs similarly on datasets that have similar meta-features. First, we aggregate the performance of Xgboost classifier across different datasets. We do this computing a simple binary metric for each dataset: whether the average performance of Xgboost is better than the average performance of all other classifiers. Then we create a visualization by running t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce the dimension of meta-features of each dataset to two, use those as x and y coordinates and color each dataset based on our newly computed metric indicating if Xgboost is better than the global average.

Figure 1 shows the outcome of this. Visually there is a fairly clear distinction, at the right bottom there is a cluster of datasets where Xgboost classifier showed sub-optimal performance compared with other models. We confirm this visual insight by training a 89.5% accurate model (naive accuracy 68.6%) that predicts, given a dataset, whether Xgboost performs better or worse than the average of other classifiers. We also look at the feature importance of the trained model, and list the top 5 important meta-features in Table 4.

| name | importance |
|---|---|
| LogInverseDatasetRatio | 0.211 |
| ClassEntropy | 0.115 |
| ClassProbabilitySTD | 0.121 |
| RatioNominalToNumerical | 0.097 |
| ClassProbabilityMean | 0.083 |

Table 4: Top 5 meta-features for Xgboost classifier

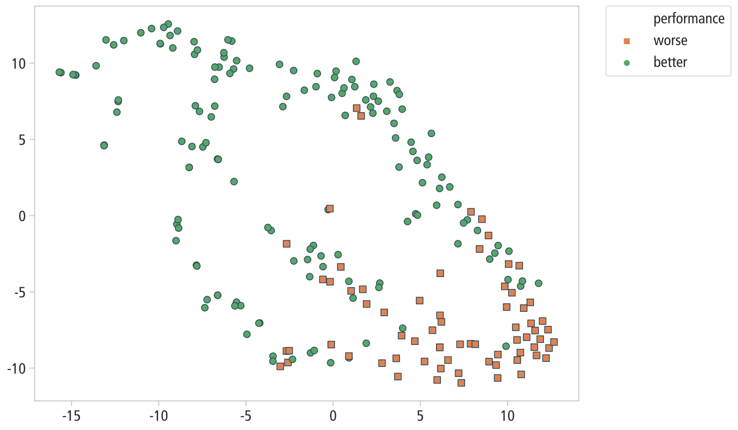

Figure 1: Plot of all datasets after running t-SNE on meta-features. Colored by if the performance of Xgboost on a given dataset is better or worse than the global average.

## F Use Case: Hyper-Parameter Optimization Simulation

Jamieson et al. propose [13] the Successive Halving algorithm for hyper-parameter optimization, which has inspired many other algoriths, e.g., Hyperband [17] and BO-HB [8]. The idea of successive halving is suggested by its name: in each round we allocate the same amount of resources (e.g., CPU) to a set of hyper-parameter configurations, evaluate the performance of all configurations, and only keep the top half. We then repeat this until only one configuration is left. However, this strategies usually involves lots of trials of training and evaluating pipelines, thus hindering fast iterations on different designs of such algorithms.

To this end, we propose to use the history of pipeline runs in Niseko to simulate the execution of hyper-parameter optimization. In other words, we use the logs of pipelines to replay the history, therefore the training and testing of pipelines is bypassed and the evaluation of hyper-parameter optimization algorithm becomes straightforward and interactive.

We are able to replicate the experiments of the original paper Successive Halving paper [13] on all 300 datasets within one hour. The average rank of successive halving, random search and random search x2 (with double resources) are respectively 2.0, 1.9, 2.3 (the higher the better), this verifies the conclusion in [13].

Furthermore, we drill down to the datasets where successive halving is worse than random search, there are 57 datasets out of 300 that successive halving is sub-optimal. We show the plot of all classification datasets after running t-SNE on meta-features in Figure 2, and we find that there is some clustering effects in the bottom left corner where successive halving is better. Also, of those 57 datasets, they tend to be small datasets, where only one dataset has more than 4,000 samples and three datasets have more than 50 features. In the future, we plan to have a deeper investigation of those datasets with Niseko's API to have a better understanding of how successive halving performs on different datasets.

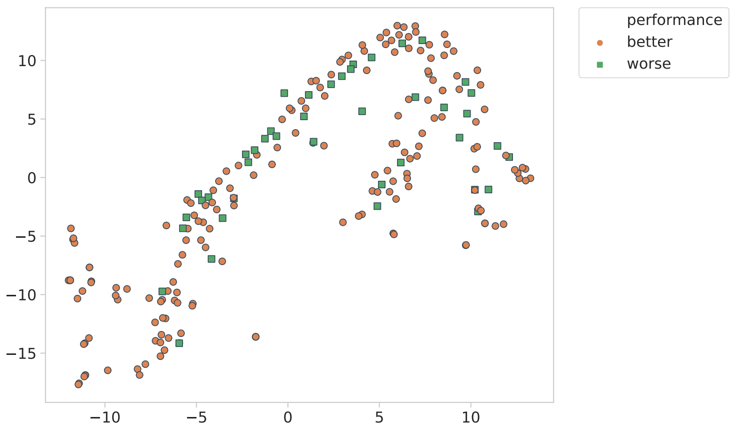

Figure 2: Plot of all datasets after running t-SNE on meta-features. Colored by if the performance of successive halving on a given dataset is better or worse than random search.

## F.1 Implementing Successive Halving in Niseko

```
import random

# create the search space
ds = context.get_dataset_by_id('185_baseball')
search_space = ds.get_pipelines()

# sample the search space
n = 128  # the number of sampled configurations
pipelines = []
for i in range(n):
    pipelines.append(random.choice(search_space))

# successive halving
round_index = 0
while len(pipelines) > 1:
    # sort by the error in this round and only keep the top half
    pipelines = sorted(pipelines, key=lambda pipeline: pipeline.
                                    pipeline_runs[round_index].
                                    error)
    pipelines = pipelines[:len(pipelines) // 2]
    round_index += 1
pipeline = pipelines[0]
print(pipeline.score)
```

The above Python snippet gives an example of implementing successive halving using Niseko's API in a couple of lines, and it takes only several seconds to finish since all computations are off-line. We are able to implement other methods as well, e.g., random search. Figure 3 exhibits the comparison between successive halving and random search using Niseko, replicating the experiments of the original paper Successive Halving paper [13].

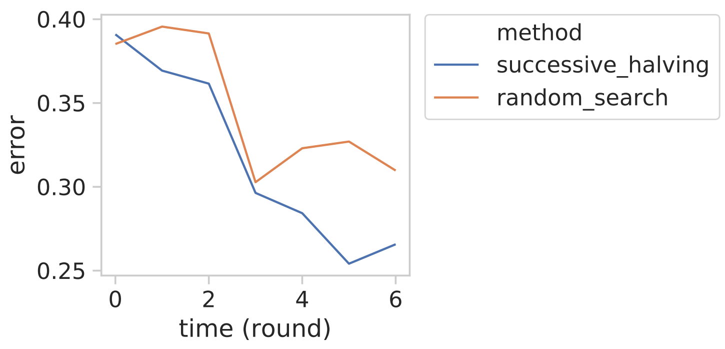

Figure 3: Comparison between successive halving and random search. The x-axis is the number of rounds, and y-axis is the error on the test split.

## G Use Case: Exploitation v.s. Exploration

When building an AutoML system, it's oftentimes important to find a good balance between exploitation (i.e., leveraging what did well in the past and searched around their neighbors) and exploration (i.e., trying out new things that have not be explored before) of pipelines or models when probing a search space. Some studies [12, 15, 2] employed acquisition function to automatically trade off exploitation and exploration, e.g., expected improvement [19]. Other studies adopted some techniques from multi-armed bandits, e.g., upper confidence bound (UCB) [6] and $\epsilon$-greedy [24].

In this section, we show that we are able to train a "agent" with Niseko using reinforcement learning to balance between exploitation and exploration, in other words, Niseko can be a gym for reinforcement learning such as OpenAI [5]. We formulate the probe of search space as a multi-armed bandit problem, where each arm is a family of models (e.g., an arm can be all SVMs and another arm can be all logistic regression models). By playing an arm, we sample a model with hyper-parameters from that arm, train it and report its performance (e.g., accuracy).

We consider this as a Markov Decision Process (MDP), and the state $s_t$ consists of the mean and standard deviation of scores for each arm, the global mean and standard deviation of scores and the current best score until round $t$. We define the reward $r(s_t, a_t, s_{t+1})$ as the difference between the previous best and the current best score in the $t$-th round with $r_0 = 0$. The cumulative reward over time is defined as:

$$R = \sum_{t=0}^{N} \gamma^t r(s_t, a_t, s_{t+1})$$

where $\gamma$ is the discount factor to favor early rewards over later rewards. We use use Q-learning [31] with function approximation to learn the action-value Q-function.

We are able to finish the experiments of comparing between reinforcement learning, random search and epsilon greedy (with $\epsilon = 0.5$) on 300 datasets within one and a half hour. The average rank of reinforcement learning, random search and epsilon greedy are respectively 2.10, 1.75, 2.15 (the higher the better), which shows that epsilon greedy has a little bit better performance than reinforcement learning. There are the 48 datasets of 300 where random search is the best among all three methods, and we plan in future work to investigate those in detail to find out the reason why random search is best on those datasets.

### G.1 Implementing Exploitation v.s. Exploration

Figure 4 compares three algorithms for balancing between exploitation and exploration on the "185_baseball" dataset. Reinforcement-learning is able to catch up quickly with epsilon greedy, and also achieves a better score in the end.

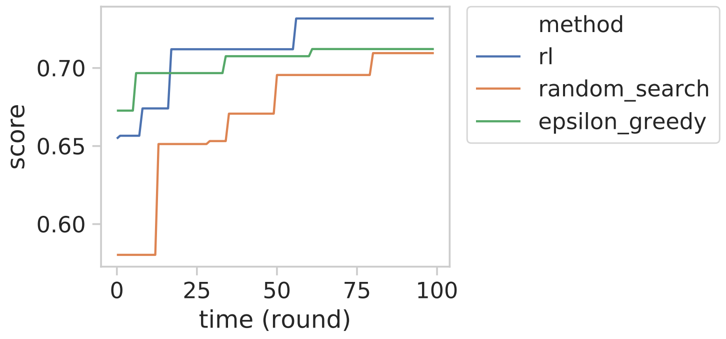

Figure 4: Comparison between reinforcement learning, random search and $\epsilon$-greedy ($\epsilon = 0.5$). The x-axis is the number of rounds, and y-axis is the score on the test split.

Since Niseko also has the full description of pipelines (e.g., what feature preprocessing methods are used), we are able to extend the search from models to feature engineering as well.

## References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[2] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[3] Pavel B Brazdil, Carlos Soares, and Joaquim Pinto Da Costa. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[4] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[6] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[7] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655, 2014.

[8] Stefan Falkner, Aaron Klein, and Frank Hutter. Combining hyperband and bayesian optimization. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS), Bayesian Optimization Workshop*, 2017.

[9] Matthias Feurer et al. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[10] Matthias Feurer et al. Initializing bayesian hyperparameter optimization via meta-learning. In *AAAI*, pages 1128–1135, 2015.

[11] Matthias Feurer, Benjamin Letham, and Eytan Bakshy. Scalable meta-learning for bayesian optimization. *arXiv preprint arXiv:1802.02219*, 2018.

[12] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[13] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[14] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[15] Neil Lawrence and Raquel Urtasun. Non-linear matrix factorization with gaussian processes. *Proceedings of the International Conference on Machine Learning*, 2009.

[16] Rui Leite and Pavel Brazdil. Predicting relative performance of classifiers from samples. In *Proceedings of the 22nd international conference on Machine learning*, pages 497–503. ACM, 2005.

[17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[18] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

[19] Matthias Schonlau, William J Welch, and Donald R Jones. Global versus local search in constrained optimization of computer models. *Lecture Notes-Monograph Series*, pages 11–25, 1998.

[20] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[21] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4077–4087, 2017.

[22] David Stern, Horst Samulowitz, Ralf Herbrich, Thore Graepel, Luca Pulina, and Armando Tacchella. Collaborative expert portfolio management. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[23] Quan Sun and Bernhard Pfahringer. Pairwise meta-rules for better meta-learning-based algorithm ranking. *Machine learning*, 93(1):141–161, 2013.

[24] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[25] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013.

[26] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[27] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-dataset: A dataset of datasets for learning to learn from few examples. *arXiv preprint arXiv:1903.03096*, 2019.

[28] Joaquin Vanschoren. Meta-learning. pages 39–68. Springer, 2018. In press, available at http://automl.org/book.

[29] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014.

[30] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial intelligence review*, 18(2):77–95, 2002.

[31] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[32] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Scalable gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107(1):43–78, 2018.

[33] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. Oboe: Collaborative filtering for automl initialization. *arXiv preprint arXiv:1808.03233*, 2018.